Workshop: Introduction to Statistical Computing with *R* and *Bioconductor*

Martin Morgan (mtmorgan@fhcrc.org), Hervé Pagès

Friday, October 4, 2013

Contents

Introduction

This hands-on workshop introduces use of *R* and *Bioconductor* for the analysis and comprehension of high-throughput genomic data. We assume basic familiarity with *R*, e.g., entering *R* commands and writing short scripts. Users will engage in exercises and activities to familiarize themselves with workshop concepts; all software is provided. The morning provides a systematic review of essential *R* data types (vector, matrix, data.frame) and programming concepts (vectorization, functions, packages). Bioinformatic data is large and complicated, and data provenance and reproducibility are important. For these reasons, we emphasize 'best practices' for writing efficient *R* code, and the use of classes and methods for working with complicated data objects. The morning highlights essential *R* packages, and provides an overview of *Bioconductor* resources for working with modern genomic data. The afternoon uses 'RNA-seq' approaches to assessing differential expression of known genes as an exemplar work flow for modern high-throughput genomic analysis. The work flow includes core activities of data manipulation, statistical analysis, and interpretation of results in biological context. The work flow provides insight into unique statistical challenges of high-throughput data, including sample normalization, use of appropriate models, approaches to maximizing information in highly structured data, and reducing the false discovery rate. Analogous statistical issues are central to many genomic data sets, and are informed by lessons learned from analysis of micro-arrays. The workshop concludes with static and interactive approaches to effective, accurate and informed presentation of statistical results in the context of interdisciplinary research teams.

Course participants will have access to a configured Amazon machine instance, with easy access through a web browser and *Rstudio*

There are many books to help with using *R*, but not yet a book-length treatment of *R* / *Bioconductor* tools for sequence analysis. Two useful references introduce the use of ranges for representing genome data in *Bioconductor* [16], and a comprehensive RNA-seq differential expression work flow [2] making extensive use of *R* / *Bioconductor* packages. Starting points for bioinformatic analysis in *R*, still relevant for statistical and informatic concepts though not directly addressing sequence analysis, are Hahne's *Bioconductor Case Studies* [11] and Gentleman's *R Programming for Bioinformatics* [10]. For *R* novices, one place to start is Pardis' *R for Beginners*[1]. General *R* programming recommendations include Dalgaard's *Introductory Statistics with R* [8], Matloff's *The Art of R Programming* [23], and Meys and de Vies' *R For Dummies* [26]; an interesting internet resource for intermediate *R* programming is Burns' *The R Inferno*[2].

A tentative agenda is in Table 1.

Table 1: Tentative agenda

Morning: R and Bioconductor

- R programming — Data types and programming concepts. Seeking help. Classes and methods.
- Bioconductor — S4 classes and methods. Essential packages for high-throughput sequencing: data representation; annotation; differential expression; ChIP-Seq; variants.
- Pitfalls and opportunities for efficient programming with large data.

Afternoon: Differential expression workflows

- Statistical issues in high-throughput genomic analysis.
- Case study: RNA-seq differential expression.
- Annotating and visualizing results.

[1] http://cran.r-project.org/doc/contrib/Paradis-rdebuts_en.pdf

[2] http://www.burns-stat.com/documents/books/the-r-inferno/

Statistical computing

Many academic and commercial software products are available; why would one use *R* and *Bioconductor*? One answer is to ask about the demands high-throughput genomic data places on effective computational biology software.

Effective computational biology software High-throughput questions make use of large data sets. This applies both to the primary data (microarray expression values, sequenced reads, etc.) and also to the annotations on those data (coordinates of genes and features such as exons or regulatory regions; participation in biological pathways, etc.). Large data sets place demands on our tools that preclude some standard approaches, such as spread sheets. Likewise, intricate relationships between data and annotation, and the diversity of research questions, require flexibility typical of a programming language rather than a narrowly-enabled graphical user interface.

Analysis of high-throughput data is necessarily statistical. The volume of data requires that it be appropriately summarized before any sort of comprehension is possible. The data are produced by advanced technologies, and these introduce artifacts (e.g., probe-specific bias in microarrays; sequence or base calling bias in RNA-seq experiments) that need to be accommodated to avoid incorrect or inefficient inference. Data sets typically derive from designed experiments, requiring a statistical approach both to account for the design and to correctly address the large number of observed values (e.g., gene expression or sequence tag counts) and small number of samples accessible in typical experiments.

Research needs to be reproducible. Reproducibility is both an ideal of the scientific method, and a pragmatic requirement. The latter comes from the long-term and multi-participant nature of contemporary science. An analysis will be performed for the initial experiment, revisited again during manuscript preparation, and revisited during reviews or in determining next steps. Likewise, analysis typically involve a team of individuals with diverse domains of expertise. Effective collaborations result when it is easy to reproduce, perhaps with minor modifications, an existing result, and when sophisticated statistical or bioinformatic analysis can be effectively conveyed to other group members.

Science moves very quickly. This is driven by the novel questions that are the hallmark of discovery, and by technological innovation and accessibility. Rapidity of scientific development places significant burdens on software, which must also move quickly. Effective software cannot be too polished, because that requires that the correct analyses are 'known' and that significant resources of time and money have been invested in developing the software; this implies software that is tracking the trailing edge of innovation. On the other hand, leading-edge software cannot be too idiosyncratic; it must be usable by a wider audience than the creator of the software, and fit in with other software relevant to the analysis.

Effective software must be accessible. Affordability is one aspect of accessibility. Another is transparent implementation, where the novel software is sufficiently documented and source code accessible enough for the assumptions, approaches, practical implementation decisions, and inevitable coding errors to be assessed by other skilled practitioners. A final aspect of affordability is that the software is actually usable. This is achieved through adequate documentation, support forums, and training opportunities.

Bioconductor as effective computational biology software What features of *R* and *Bioconductor* contribute to its effectiveness as a software tool?

Bioconductor is well suited to handle extensive data and annotation. *Bioconductor* 'classes' represent high-throughput data and their annotation in an integrated way. *Bioconductor* methods use advanced programming techniques or *R* resources (such as transparent data base or network access) to minimize memory requirements and integrate with diverse resources. Classes and methods coordinate complicated data sets with extensive annotation. Nonetheless, the basic model for object manipulation in *R* involves vectorized in-memory representations. For this reason, particular programming paradigms (e.g., block processing of data streams; explicit parallelism) or hardware resources (e.g., large-memory computers) are sometimes required when dealing with extensive data.

R is ideally suited to addressing the statistical challenges of high-throughput data. Three examples include the development of the 'RMA' and other normalization algorithm for microarray pre-processing, use of moderated t-statistics for assessing microarray differential expression, and development of negative binomial approaches to estimating dispersion read counts necessary for appropriate analysis of RNAseq designed experiments.

Many of the 'old school' aspects of *R* and *Bioconductor* facilitate reproducible research. An analysis is often represented as a text-based script. Reproducing the analysis involves re-running the script; adjusting how the analysis is performed involves simple text-editing tasks. Beyond this, *R* has the notion of a 'vignette', which represents an analysis as a LaTeX document with embedded *R* commands. The *R* commands are evaluated when the document is built, thus reproducing the analysis. The use of LaTeX means that the symbolic manipulations in the script are augmented with textual explanations and justifications for the approach taken; these include graphical and tabular summaries at appropriate places in the analysis. *R* includes facilities for reporting the exact version of *R* and associated packages used in an analysis so that, if needed, discrepancies between software versions can be tracked down and their importance evaluated. While users often think of *R* packages as providing new functionality, packages are also used to enhance reproducibility by encapsulating a single analysis. The package can contain data sets, vignette(s) describing the analysis, *R* functions that might have been written, scripts for key data processing stages, and documentation (via standard *R* help mechanisms) of what the functions, data, and packages are about.

The *Bioconductor* project adopts practices that facilitate reproducibility. Versions of *Bioconductor* are released twice each year. Each *Bioconductor* release is the result of development, in a separate branch, during the previous six months. The release is built daily against the corresponding version of *R* on Linux, Mac, and Windows platforms, with an extensive suite of tests performed. The `biocLite` function ensures that each release of *R* uses the corresponding *Bioconductor* packages. The user thus has access to stable and tested package versions. *R* and *Bioconductor* are effective tools for reproducible research.

R and *Bioconductor* exist on the leading portion of the software life cycle. Contributors are primarily from academic institutions, and are directly involved in novel research activities. New developments are made available in a familiar format, i.e., the *R* language, packaging, and build systems. The rich set of facilities in *R* (e.g., for advanced statistical analysis or visualization) and the extensive resources in *Bioconductor* (e.g., for annotation using third-party data such as Biomart or UCSC genome browser tracks) mean that innovations can be directly incorporated into existing work flows. The 'development' branches of *R* and *Bioconductor* provide an environment where contributors can explore new approaches without alienating their user base.

R and *Bioconductor* also fair well in terms of accessibility. The software is freely available. The source code is easily and fully accessible for critical evaluation. The *R* packaging and check system requires that all functions are documented. *Bioconductor* requires that each package contain vignettes to illustrate the use of the software. There are very active *R* and *Bioconductor* mailing lists for immediate support, and regular training and conference activities for professional development.

Part I

R / Bioconductor

Chapter 1

Basics of R

1.1 Statistical analysis and comprehension

R is an open-source statistical programming language. It is used to manipulate data, to perform statistical analysis, and to present graphical and other results. *R* consists of a core language, additional 'packages' distributed with the *R* language, and a very large number of packages contributed by the broader community. Packages add specific functionality to an *R* installation. *R* has become the primary language of academic statistical analysis, and is widely used in diverse areas of research, government, and industry.

R has several unique features. It has a surprisingly 'old school' interface: users type commands into a console; scripts in plain text represent work flows; tools other than *R* are used for editing and other tasks. *R* is a flexible programming language, so while one person might use functions provided by *R* to accomplish advanced analytic tasks, another might implement their own functions for novel data types. As a programming language, *R* adopts syntax and grammar that differ from many other languages: objects in *R* are 'vectors', and functions are 'vectorized' to operate on all elements of the object; *R* objects have 'copy on change' and 'pass by value' semantics, reducing unexpected consequences for users at the expense of less efficient memory use; common paradigms in other languages, such as the 'for' loop, are encountered much less commonly in *R*. Many authors contribute to *R*, so there can be a frustrating inconsistency of documentation and interface. *R* grew up in the academic community, so authors have not shied away from trying new approaches. Common statistical analysis functions are very well-developed.

Opening an *R* session results in a prompt. The user types instructions at the prompt. Here is an example:

```
> ## assign values 5, 4, 3, 2, 1 to variable 'x'
> x <- c(5, 4, 3, 2, 1)
> x

[1] 5 4 3 2 1
```

The first line starts with a # to represent a comment; the line is ignored by *R*. The next line creates a variable `x`. The variable is assigned (using <-, we could have used = almost interchangeably) a value. The value assigned is the result of a call to the c function. That it is a function call is indicated by the symbol named followed by parentheses, `c()`. The c function takes zero or more arguments, and returns a vector. The vector is the value assigned to `x`. *R* responds to this line with a new prompt, ready for the next input. The next line asks *R* to display the value of the variable `x`. *R* responds by printing `[1]` to indicate that the subsequent number is the first element of the vector. It then prints the value of `x`.

R has many features to aid common operations. Entering sequences is a very common operation, and expressions of the form `2:4` create a sequence from 2 to 4. Sub-setting one vector by another is enabled with `[`. Here we create an integer sequence from 2 to 4, and use the sequence as an index to select the second, third, and fourth elements of `x`

```
> x[2:4]

[1] 4 3 2
```

Index values can be repeated, and if outside the domain of `x` return the special value `NA`. Negative index values remove elements from the vector. Logical and character vectors (described below) can also be used for sub-setting.

Table 1.1: Essential aspects of the *R* language.

Category	Function	Description

Vectors	`integer`, `numeric`
	`complex`, `character`
	`raw`, `factor`
		Vectors of length $>= 0$ holding a single data type

List-like	`list`	Arbitrary collections of elements
	`data.frame`	List of equal-length vectors
	`environment`	*Pass-by-reference* data storage; hash
Array-like	`data.frame`	Homogeneous columns; row- and column indexing
	`array`	0 or more dimensions
	`matrix`	Two-dimensional, homogeneous types
Statistical	`NA`, `factor`	Essential statistical concepts, integral to the language.
Classes	'S3'	List-like structured data; simple inheritance & dispatch
	'S4'	Formal classes, multiple inheritance & dispatch
Functions	'function'	A simple function with arguments, body, and return value
	'generic'	A (S3 or S4) function with associated *methods*
	'method'	A function implementing a generic for an S3 or S4 class

R functions operate on variables. Functions are usually vectorized, acting on all elements of their argument and obviating the need for explicit iteration. Functions can generate warnings when performing suspect operations, or errors if evaluation cannot proceed; try `log(-1)`.

```
> log(x)

[1] 1.6094379 1.3862944 1.0986123 0.6931472 0.0000000
```

1.2 Essential data types

R has a number of built-in data types, summarized in Table 1.1. These represent `integer`, `numeric` (floating point), `complex`, `character`, `logical` (Boolean), and `raw` (byte) data. It is possible to convert between data types, and to discover the type or mode of a variable.

```
> c(1.1, 1.2, 1.3)         # numeric

[1] 1.1 1.2 1.3

> c(FALSE, TRUE, FALSE)    # logical

[1] FALSE  TRUE FALSE

> c("foo", "bar", "baz")   # character, single or double quote ok

[1] "foo" "bar" "baz"

> as.character(x)          # convert 'x' to character

[1] "5" "4" "3" "2" "1"

> typeof(x)                # the number 5 is numeric, not integer

[1] "double"

> typeof(2L)               # append 'L' to force integer

[1] "integer"
```

```
> typeof(2:4)              # ':' produces a sequence of integers
```

```
[1] "integer"
```

R includes data types particularly useful for statistical analysis, including `factor` to represent categories and `NA` (used in any vector) to represent missing values.

```
> sex <- factor(c("Male", "Female", NA), levels=c("Female", "Male"))
> sex
```

```
[1] Male   Female <NA>
Levels: Female Male
```

Lists, data frames, and matrices All of the vectors mentioned so far are homogeneous, consisting of a single type of element. A `list` can contain a collection of different types of elements and, like all vectors, these elements can be named to create a key-value association.

```
> lst <- list(a=1:3, b=c("foo", "bar"), c=sex)
> lst
```

```
$a
[1] 1 2 3

$b
[1] "foo" "bar"

$c
[1] Male   Female <NA>
Levels: Female Male
```

Lists can be subset like other vectors to get another list, or subset with `[[` to retrieve the actual list element; as with other vectors, sub-setting can use names

```
> lst[c(3, 1)]             # another list -- class isomorphism
```

```
$c
[1] Male   Female <NA>
Levels: Female Male

$a
[1] 1 2 3
```

```
> lst[["a"]]               # the element itself, selected by name
```

```
[1] 1 2 3
```

A `data.frame` is a list of equal-length vectors, representing a rectangular data structure not unlike a spread sheet. Each column of the data frame is a vector, so data types must be homogeneous within a column. A `data.frame` can be subset by row or column, and columns can be accessed with `$` or `[[`.

```
> df <- data.frame(age=c(27L, 32L, 19L),
+                  sex=factor(c("Male", "Female", "Male")))
> df
```

```
  age    sex
1  27   Male
2  32 Female
3  19   Male
```

```
> df[c(1, 3),]

  age  sex
1  27 Male
3  19 Male

> df[df$age > 20,]

  age    sex
1  27   Male
2  32 Female
```

A `matrix` is also a rectangular data structure, but subject to the constraint that all elements are the same type. A matrix is created by taking a vector, and specifying the number of rows or columns the vector is to represent.

```
> m <- matrix(1:12, nrow=3)
> m

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

> m[c(1, 3), c(2, 4)]

     [,1] [,2]
[1,]    4   10
[2,]    6   12
```

On sub-setting, *R* coerces a single column `data.frame` or single row or column `matrix` to a vector if possible; use `drop=FALSE` to stop this behavior.

```
> m[, 3]

[1] 7 8 9

> m[, 3, drop=FALSE]

     [,1]
[1,]    7
[2,]    8
[3,]    9
```

An `array` is a data structure for representing homogeneous, rectangular data in higher dimensions.

1.3 S3 (and S4) classes

More complicated data structures are represented using the 'S3' or 'S4' object system. Objects are often created by functions (for example, `lm`, below), with parts of the object extracted or assigned using *accessor* functions. The following generates 1000 random normal deviates as `x`, and uses these to create another 1000 deviates `y` that are linearly related to `x` but with some error. We fit a linear regression using a 'formula' to describe the relationship between variables, summarize the results in a familiar ANOVA table, and access `fit` (an S3 object) for the residuals of the regression, using these as input first to the `var` (variance) and then `sqrt` (square-root) functions. Objects can be interrogated for their class.

```
> x <- rnorm(1000, sd=1)
> y <- x + rnorm(1000, sd=.5)
> fit <- lm(y ~ x)        # formula describes linear regression
> fit                     # an 'S3' object
```

```
Call:
lm(formula = y ~ x)

Coefficients:
(Intercept)            x
    0.00995      0.98005

> anova(fit)

Analysis of Variance Table

Response: y
           Df  Sum Sq Mean Sq F value    Pr(>F)
x           1 1027.48 1027.48  3999.2 < 2.2e-16 ***
Residuals 998  256.41    0.26
---
Signif. codes:  0 âĂŸ***âĂŹ 0.001 âĂŸ**âĂŹ 0.01 âĂŸ*âĂŹ 0.05 âĂŸ.âĂŹ 0.1 âĂŸ âĂŹ 1

> sqrt(var(resid(fit)))  # residuals accessor and subsequent transforms

[1] 0.5066236

> class(fit)

[1] "lm"
```

Many *Bioconductor* packages implement S4 objects to represent data. S3 and S4 systems are quite different from a programmer's perspective, but fairly similar from a user's perspective: both systems encapsulate complicated data structures, and allow for methods specialized to different data types; accessors are used to extract information from the objects.

1.4 Functions

R has a very large number of functions; Table 1.2 provides a brief list of those that might be commonly used and particularly useful. See the help pages (e.g., `?lm`) and examples (`example(match)`) for more details on each of these functions.

R functions accept arguments, and return values. Arguments can be required or optional. Some functions may take variable numbers of arguments, e.g., the columns in a `data.frame`

```
> y <- 5:1
> log(y)

[1] 1.6094379 1.3862944 1.0986123 0.6931472 0.0000000

> args(log)        # arguments 'x' and 'base'; see ?log

function (x, base = exp(1))
NULL

> log(y, base=2)   # 'base' is optional, with default value

[1] 2.321928 2.000000 1.584963 1.000000 0.000000

> try(log())       # 'x' required; 'try' continues even on error
> args(data.frame) # ... represents variable number of arguments
```

Table 1.2: A selection of *R* function.

dir, read.table **(and friends),** scan List files in a directory, read spreadsheet-like data into *R*, efficiently read homogeneous data (e.g., a file of numeric values) to be represented as a matrix.

c, factor, data.frame, matrix Create a vector, factor, data frame or matrix.

summary, table, xtabs Summarize, create a table of the number of times elements occur in a vector, cross-tabulate two or more variables.

t.test, aov, lm, anova, chisq.test Basic comparison of two (t.test) groups, or several groups via analysis of variance / linear models (aov output is probably more familiar to biologists), or compare simpler with more complicated models (anova); χ^2 tests.

dist, hclust Cluster data.

plot Plot data.

ls, str, library, search List objects in the current (or specified) workspace, or peak at the structure of an object; add a library to or describe the search path of attached packages.

lapply, sapply, mapply, aggregate Apply a function to each element of a list (lapply, sapply), to elements of several lists (mapply), or to elements of a list partitioned by one or more factors (aggregate).

with Conveniently access columns of a data frame or other element without having to repeat the name of the data frame.

match, %in% Report the index or existence of elements from one vector that match another.

split, cut, unlist Split one vector by an equal length factor, cut a single vector into intervals encoded as levels of a factor, unlist (concatenate) list elements.

strsplit, grep, sub Operate on character vectors, splitting it into distinct fields, searching for the occurrence of a patterns using regular expressions (see ?regex, or substituting a string for a regular expression.

biocLite, install.packages Install a package from an on-line repository into your *R*.

traceback, debug, browser Report the sequence of functions under evaluation at the time of the error; enter a debugger when a particular function or statement is invoked.

```
function (..., row.names = NULL, check.rows = FALSE, check.names = TRUE,
    stringsAsFactors = default.stringsAsFactors())
NULL
```

Arguments can be matched by name or position. If an argument appears after ..., it must be named.

```
> log(base=2, y)   # match argument 'base' by name, 'x' by position
```

```
[1] 2.321928 2.000000 1.584963 1.000000 0.000000
```

A function such as anova is a *generic* that provides an overall signature but dispatches the actual work to the *method* corresponding to the class(es) of the arguments used to invoke the generic. A generic may have fewer arguments than a method, as with the S3 function anova and its method anova.glm.

```
> args(anova)
```

```
function (object, ...)
NULL
```

```
> args(anova.glm)
```

```
function (object, ..., dispersion = NULL, test = NULL)
NULL
```

The ... argument in the anova generic means that additional arguments are possible; the anova generic hands these arguments to the method it dispatches to.

Table 1.3: Tools for debugging and error-handling.

Function	Description
`traceback`	Report the 'call stack' at the time of an error.
`options(error=)`	Set a handler to be executed on error, e.g., `error=recover`.
`debug`, `trace`	Enter the browser when a function is called
`browser`	Interactive debugging
`tryCatch`	Handle an error condition in a script.

1.5 Warnings, errors, and debugging

R signals unexpected results through warnings and errors. Warnings occur when the calculation produces an unusual result that nonetheless does not preclude further evaluation. For instance `log(-1)` results in a value `NaN` ('not a number') that allows computation to continue, but at the same time signals a warning

```
> log(-1)
[1] NaN
Warning message:
In log(-1) : NaNs produced
```

Errors result when the inputs or outputs of a function are such that no further action can be taken, e.g., trying to take the square root of a character vector

```
> sqrt("two")
Error in sqrt("two") : Non-numeric argument to mathematical function
```

Warnings and errors occurring at the command prompt are usually easy to diagnose. They can be more enigmatic when occurring in a function, and exacerbated by sometimes cryptic (when read out of context) error messages. Some key tools for figuring out ('debugging') errors are summarized in Table 1.3.

An initial step in coming to terms with errors is to simplify the problem as much as possible, aiming for a 'reproducible' error. The reproducible error might involve a very small (even trivial) data set that immediately provokes the error. Often the process of creating a reproducible example helps to clarify what the error is, and what possible solutions might be.

Invoking `traceback()` immediately after an error occurs provides a 'stack' of the function calls that were in effect when the error occurred. This can help understand the context in which the error occurred. Knowing the context, one might use `debug` (or its more elaborate cousin, `trace`) to enter into a browser (see `?browser`) that allows one to step through the function in which the error occurred.

It can sometimes be useful to use global options (see `?options`) to influence what happens when an error occurs. Two common global options are `error` and `warn`. Setting `error=recover` combines the functionality of `traceback` and `debug`, allowing the user to enter the browser at any level of the call stack in effect at the time the error occurred. Default error behavior can be restored with `options(error=NULL)`. Setting `warn=2` causes warnings to be promoted to errors. For instance, initial investigation of an error might show that the error occurs when one of the arguments to a function has value `NaN`. The error might be accompanied by a warning message that the `NaN` has been introduced, but because warnings are by default not reported immediately it is not clear where the `NaN` comes from. `warn=2` means that the warning is treated as an error, and hence can be debugged using `traceback`, `debug`, and so on.

It is possible to continue evaluation even after an error occurs. The simplest mechanism uses the `try` function, but an only slightly more complicated version providing greater flexibility is `tryCatch`. `tryCatch` allows one to write a `handler` (the `error` argument to `tryCatch`, below) to address common faults in a way that allows a script to continue executing. Suppose a function `f` fails under certain conditions

```
> f <- function(i) {
+     if (i < 0)
+         stop("i is negative")
+     rnorm(i)
+ }
> lapply(0:1, f)
```

Table 1.4: Selected base and contributed packages.

Package	Description
base	Data input and manipulation; scripting and programming.
stats	Essential statistical and plotting functions.
lattice, *ggplot2*	Approaches to advanced graphics.
methods	'S4' classes and methods.
parallel	Facilities for parallel evaluation.
Matrix	Diverse matrix representations
data.table	Efficient management of large data tables

```
[[1]]
numeric(0)

[[2]]
[1] 1.024123
```

but we wish to continue, e.g., replacing failed conditions with NA:

```
> lapply(-1:1, function(i) {
+     tryCatch({
+         f(i)
+     }, error=function(err) {
+         ## return 'NA' when error occurs, instead of stopping
+         NA_real_
+     })
+ })

[[1]]
[1] NA

[[2]]
numeric(0)

[[3]]
[1] -1.594477
```

1.6 Packages

Packages provide functionality beyond that available in base *R*. There are over 4000 packages in CRAN (comprehensive *R* archive network) and more than 670 *Bioconductor* packages. Packages are contributed by diverse members of the community; they vary in quality (many are excellent) and sometimes contain idiosyncratic aspects to their implementation. Table 1.4 outlines key base packages and selected contributed packages; see a local CRAN mirror (including the task views summarizing packages in different domains) and *Bioconductor* for additional contributed packages.

The *lattice* package illustrates the value packages add to base *R*. *lattice* is distributed with *R* but not loaded by default. It provides a very expressive way to visualize data. The following example plots yield for a number of barley varieties, conditioned on site and grouped by year. Figure 1.1 is read from the lower left corner. Note the common scales, efficient use of space, and not-too-pleasing default color palette. The Morris sample appears to be mis-labeled for 'year', an apparent error in the original data. Find out about the built-in data set used in this example with ?barley.

```
> library(lattice)
> plt <- dotplot(variety ~ yield | site, data = barley, groups = year,
+                xlab = "Barley Yield (bushels/acre)" , ylab=NULL,
+                key = simpleKey(levels(barley$year), space = "top",
```

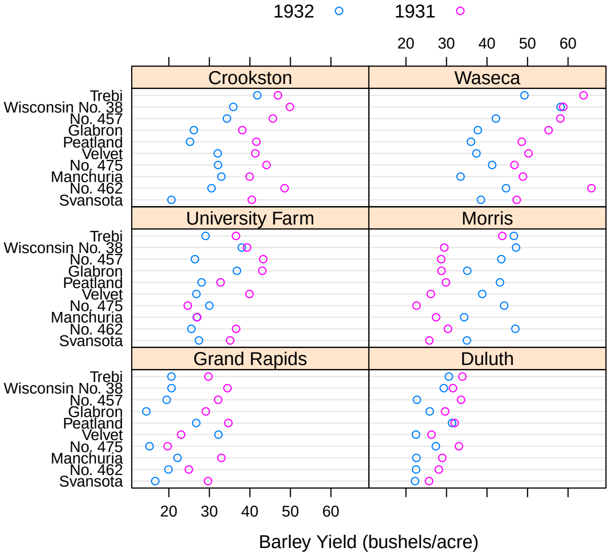


Figure 1.1: Variety yield conditional on site and grouped by year, for the `barley` data set. What's up with Morris?

```
+                   columns=2),
+                 aspect=0.5, layout = c(2,3))
> print(plt)
```

New packages (from *Bioconductor* or CRAN) can be added to an *R* installation using `biocLite()`:

```
source("http://bioconductor.org/biocLite.R")
biocLite(c("GenomicRanges", "ShortRead"))
```

A package is installed only once per *R* installation, but needs to be loaded (with `library`) in each session in which it is used. Loading a package also loads any package that it depends on. Packages loaded in the current session are displayed with `search`. The ordering of packages returned by `search` represents the order in which the global environment (where commands entered at the prompt are evaluated) and attached packages are searched for symbols.

```
> length(search())
```

```
[1] 21
```

```
> search()
```

```
 [1] ".GlobalEnv"                      "package:lattice"
 [3] "package:SequenceAnalysisData" "package:edgeR"
 [5] "package:limma"                   "package:GenomicFeatures"
 [7] "package:AnnotationDbi"           "package:Biobase"
 [9] "package:GenomicRanges"           "package:XVector"
[11] "package:IRanges"                 "package:BiocGenerics"
[13] "package:parallel"                "package:stats"
[15] "package:graphics"                "package:grDevices"
[17] "package:utils"                   "package:datasets"
[19] "package:methods"                 "Autoloads"
[21] "package:base"
```

It is possible for a package earlier in the search path to mask symbols later in the search path; these can be disambiguated using `::`.

```
> pi <- 3.2     ## http://en.wikipedia.org/wiki/Indiana_Pi_Bill
> base::pi

[1] 3.141593

> rm(pi)        ## remove from the .GlobalEnv
```

Exercise 1
Use the `library` *function to load the StatisticalComputing2013 package. Use the* `sessionInfo` *function to verify that you are using R version 3.0.2 and current packages, similar to those reported here. What other packages were loaded along with StatisticalComputing2013?*

Solution:

```
> library(StatisticalComputing2013)
> sessionInfo()
```

1.7 Help!

Find help using the *R* help system. Start a web browser with

```
> help.start()
```

The 'Search Engine and Keywords' link is helpful in day-to-day use.

Manual pages Use manual pages to find detailed descriptions of the arguments and return values of functions, and the structure and methods of classes. Find help within an *R* session as

```
> ?data.frame
> ?lm
> ?anova             # a generic function
> ?anova.lm          # an S3 method, specialized for 'lm' objects
```

S3 methods can be queried interactively. For S3,

```
> methods(anova)

[1] anova.MAList  anova.glm     anova.glmlist anova.lm      anova.loess*
[6] anova.mlm     anova.nls*

   Non-visible functions are asterisked

> methods(class="glm")

 [1] add1.glm*           anova.glm           confint.glm*
 [4] cooks.distance.glm* deviance.glm*       drop1.glm*
 [7] effects.glm*        extractAIC.glm*     family.glm*
[10] formula.glm*        influence.glm*      logLik.glm*
[13] model.frame.glm     nobs.glm*           predict.glm
[16] print.glm           residuals.glm       rstandard.glm
[19] rstudent.glm        summary.glm         vcov.glm*
[22] weights.glm*

   Non-visible functions are asterisked
```

It is often useful to view a method definition, either by typing the method name at the command line or, for 'non-visible' methods, using getAnywhere:

```
> anova.lm
> getAnywhere("anova.loess")
```

For instance, the source code of a function is printed if the function is invoked without parentheses. Here we discover that the function head (which returns the first 6 elements of anything) defined in the *utils* package, is an S3 generic (indicated by UseMethod) and has several methods. We use head to look at the first six lines of the head method specialized for matrix objects.

```
> utils::head

function (x, ...)
UseMethod("head")
<bytecode: 0x48a98d8>
<environment: namespace:utils>

> methods(head)

[1] head.Vector      head.data.frame* head.default*    head.ftable*
[5] head.function*   head.matrix      head.table*

   Non-visible functions are asterisked

> head(head.matrix)

1 function (x, n = 6L, ...)
2 {
3     stopifnot(length(n) == 1L)
4     n <- if (n < 0L)
5         max(nrow(x) + n, 0L)
6     else min(n, nrow(x))
```

Vignettes Vignettes, especially in *Bioconductor* packages, provide an extensive narrative describing overall package functionality. Use

```
> vignette(package="StatisticalComputing2013")
```

to see, in your web browser, vignettes available in the *StatisticalComputing2013* package. Vignettes usually consist of text with embedded *R* code, a form of literate programming. The vignette can be read as a PDF document, while the *R* source code is present as a script file ending with extension .R. The script file can be sourced or copied into an *R* session to evaluate exactly the commands used in the vignette. For *Bioconductor* packages, vignettes are available on the package 'landing page', e.g., for *IRanges*[1]

1.8 Exercises

Exercise 2
This exercise uses data describing 128 microarray samples as a basis for exploring R functions. Covariates such as age, sex, type, stage of the disease, etc., are in a data file pData.csv*.*

The following command creates a variable pdataFiles *that is the location of a comma-separated value ('csv') file to be used in the exercise. A csv file can be created using, e.g., 'Save as...' in spreadsheet software.*

```
> pdataFile <- system.file(package="SequenceAnalysisData", "extdata",
+                          "pData.csv")
```

[1] http://bioconductor.org/packages/devel/bioc/html/IRanges.html

Input the csv file using `read.table`*, assigning the input to a variable* `pdata`*. Use* `dim` *to find out the dimensions (number of rows, number of columns) in the object. Are there 128 rows? Use* `names` *or* `colnames` *to list the names of the columns of* `pdata`*. Use* `summary` *to summarize each column of the data. What are the data types of each column in the data frame?*

A data frame is a list of equal length vectors. Select the 'sex' column of the data frame using `[[` *or* `$`*. Pause to explain to your neighbor why this sub-setting works. Since a data frame is a list, use* `sapply` *to ask about the class of each column in the data frame. Explain to your neighbor what this produces, and why.*

Use `table` *to summarize the number of males and females in the sample. Consult the help page* `?table` *to figure out additional arguments required to include* `NA` *values in the tabulation.*

The `mol.biol` *column summarizes molecular biological attributes of each sample. Use* `table` *to summarize the different molecular biology levels in the sample. Use* `%in%` *to create a logical vector of the samples that are either* `BCR/ABL` *or* `NEG`*. Subset the original phenotypic data to contain those samples that are* `BCR/ABL` *or* `NEG`*.*

After sub-setting, what are the levels of the `mol.biol` *column? Set the levels to be* `BCR/ABL` *and* `NEG`*, i.e., the levels in the subset.*

One would like covariates to be similar across groups of interest. Use `t.test` *to assess whether* `BCR/ABL` *and* `NEG` *have individuals with similar age. To do this, use a* `formula` *that describes the response* `age` *in terms of the predictor* `mol.biol`*. If age is not independent of molecular biology, what complications might this introduce into subsequent analysis? Use the* `boxplot` *function to visualize the relationship between* `age` *and* `mol.biol`*.*

Solution: Here we input the data and explore basic properties.

```
> pdata <- read.table(pdataFile)
> dim(pdata)
```

```
[1] 128  21
```

```
> names(pdata)
```

```
 [1] "cod"            "diagnosis"      "sex"            "age"           
 [5] "BT"             "remission"      "CR"             "date.cr"       
 [9] "t.4.11."        "t.9.22."        "cyto.normal"    "citog"         
[13] "mol.biol"       "fusion.protein" "mdr"            "kinet"         
[17] "ccr"            "relapse"        "transplant"     "f.u"           
[21] "date.last.seen"
```

```
> summary(pdata)
```

```
      cod          diagnosis     sex          age             BT
 10005  :  1   1/15/1997 :  2   F   :42   Min.   : 5.00   B2     :36
 1003   :  1   1/29/1997 :  2   M   :83   1st Qu.:19.00   B3     :23
 1005   :  1   11/15/1997:  2   NA's: 3   Median :29.00   B1     :19
 1007   :  1   2/10/1998 :  2             Mean   :32.37   T2     :15
 1010   :  1   2/10/2000 :  2             3rd Qu.:45.50   B4     :12
 11002  :  1   (Other)   :116             Max.   :58.00   T3     :10
 (Other):122   NA's      :  2             NA's   :5       (Other):13
 remission                  CR           date.cr    t.4.11.
 CR  :99   CR                :96   11/11/1997: 3   Mode :logical
 REF :15   DEATH IN CR       : 3   1/21/1998 : 2   FALSE:86
 NA's:14   DEATH IN INDUCTION: 7   10/18/1999: 2   TRUE :7
           REF               :15   12/7/1998 : 2   NA's :35
           NA's              : 7   1/17/1997 : 1
                                   (Other)   :87
                                   NA's      :31
  t.9.22.        cyto.normal               citog        mol.biol
 Mode :logical   Mode :logical   normal       :24   ALL1/AF4:10
 FALSE:67        FALSE:69        simple alt.  :15   BCR/ABL :37
 TRUE :26        TRUE :24        t(9;22)      :12   E2A/PBX1: 5  
 NA's :35        NA's :35        t(9;22)+other:11   NEG     :74  
                                 complex alt. :10   NUP-98  : 1  
                                 (Other)      :21   p15/p16 : 1  
                                 NA's         :35                
   fusion.protein   mdr          kinet       ccr           relapse      
 p190     :17     NEG :101   dyploid:94   Mode :logical   Mode :logical  
 p190/p210: 8     POS : 24   hyperd.:27   FALSE:74        FALSE:35       
 p210     : 8     NA's:  3   NA's   : 7   TRUE :26        TRUE :65       
 NA's     :95                             NA's :28        NA's :28       
```

```
 transplant                     f.u        date.last.seen
 Mode :logical   REL              :61   1/7/1998  : 2  
 FALSE:91        CCR              :23   12/15/1997: 2  
 TRUE :9         BMT / DEATH IN CR: 4   12/31/2002: 2  
 NA's :28        BMT / CCR        : 3   3/29/2001 : 2  
                 DEATH IN CR      : 2   7/11/1997 : 2  
                 (Other)          : 7   (Other)   :83  
                 NA's             :28   NA's      :35  
```

A data frame can be subset as if it were a matrix, or a list of column vectors.

```
> head(pdata[,"sex"], 3)
```

```
[1] M M F
Levels: F M
```

```
> head(pdata$sex, 3)
```

```
[1] M M F
Levels: F M
```

```
> head(pdata[["sex"]], 3)
```

```
[1] M M F
Levels: F M
```

```
> sapply(pdata, class)
```

```
           cod      diagnosis            sex            age             BT 
      "factor"       "factor"       "factor"      "integer"       "factor" 
     remission             CR        date.cr        t.4.11.        t.9.22. 
      "factor"       "factor"       "factor"      "logical"      "logical" 
   cyto.normal          citog       mol.biol fusion.protein            mdr 
     "logical"       "factor"       "factor"       "factor"       "factor" 
         kinet            ccr        relapse     transplant            f.u 
      "factor"      "logical"      "logical"      "logical"       "factor" 
date.last.seen 
      "factor" 
```

The number of males and females, including NA, is

```
> table(pdata$sex, useNA="ifany")
```

```
   F    M <NA> 
  42   83    3 
```

An alternative version of this uses the `with` function: `with(pdata, table(sex, useNA="ifany"))`.

The `mol.biol` column contains the following samples:

```
> with(pdata, table(mol.biol, useNA="ifany"))
```

```
mol.biol
ALL1/AF4  BCR/ABL E2A/PBX1      NEG   NUP-98  p15/p16
      10       37        5       74        1        1
```

A logical vector indicating that the corresponding row is either `BCR/ABL` or `NEG` is constructed as

```
> ridx <- pdata$mol.biol %in% c("BCR/ABL", "NEG")
```

We can get a sense of the number of rows selected via `table` or `sum` (discuss with your neighbor what `sum` does, and why the answer is the same as the number of `TRUE` values in the result of the `table` function).

```
> table(ridx)
```

```
ridx
FALSE  TRUE
   17   111
```

```
> sum(ridx)
```

```
[1] 111
```

The original data frame can be subset to contain only `BCR/ABL` or `NEG` samples using the logical vector `ridx` that we created.

```
> pdata1 <- pdata[ridx,]
```

The levels of each factor reflect the levels in the original object, rather than the levels in the subset object, e.g.,

```
> levels(pdata1$mol.biol)
```

```
[1] "ALL1/AF4" "BCR/ABL"  "E2A/PBX1" "NEG"      "NUP-98"   "p15/p16"
```

These can be re-coded by updating the new data frame to contain a factor with the desired levels.

```
> pdata1$mol.biol <- factor(pdata1$mol.biol)
> table(pdata1$mol.biol)
```

```
BCR/ABL     NEG
     37      74
```

To ask whether age differs between molecular biology types, we use a formula `age ~ mol.biol` to describe the relationship ('age as a function of molecular biology') that we wish to test

```
> with(pdata1, t.test(age ~ mol.biol))
```

```
        Welch Two Sample t-test

data:  age by mol.biol
t = 4.8172, df = 68.529, p-value = 8.401e-06
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
  7.13507 17.22408
sample estimates:
mean in group BCR/ABL     mean in group NEG
             40.25000              28.07042
```

This summary can be visualize with, e.g., the `boxplot` function

```
> ## not evaluated
> boxplot(age ~ mol.biol, pdata1)
```

Molecular biology seem to be strongly associated with age; individuals in the `NEG` group are considerably younger than those in the `BCR/ABL` group. We might wish to include age as a covariate in any subsequent analysis seeking to relate molecular biology to gene expression.

Chapter 2

Bioconductor

Bioconductor is a collection of *R* packages for the analysis and comprehension of high-throughput genomic data. *Bioconductor* started more than 10 years ago, and is widely used (Figure 2.1). It gained credibility for its statistically rigorous approach to microarray pre-processing and analysis of designed experiments, and integrative and reproducible approaches to bioinformatic tasks. There are now more than 670 *Bioconductor* packages for expression and other microarrays, sequence analysis, flow cytometry, imaging, and other domains. The *Bioconductor* web site provides installation, package repository, help, and other documentation.

The *Bioconductor* web site is at `bioconductor.org`. Features include:

- Introductory work flows.
- A manifest of *Bioconductor* packages arranged in BiocViews.
- Annotation (data bases of relevant genomic information, e.g., Entrez gene ids in model organisms, KEGG pathways) and experiment data (containing relatively comprehensive data sets and their analysis) packages.
- Mailing lists, including searchable archives, as the primary source of help.
- Course and conference information, including extensive reference material.
- General information about the project.
- Package developer resources, including guidelines for creating and submitting new packages.

Exercise 3

Scavenger hunt. Spend five minutes tracking down the following information.

a. From the Bioconductor web site, instructions for installing or updating Bioconductor packages.

b. A list of all packages in the current release of Bioconductor.

c. The URL of the Bioconductor mailing list subscription page.

Solution: Possible solutions from the *Bioconductor* web site are, e.g., `http://bioconductor.org/install/` (in-

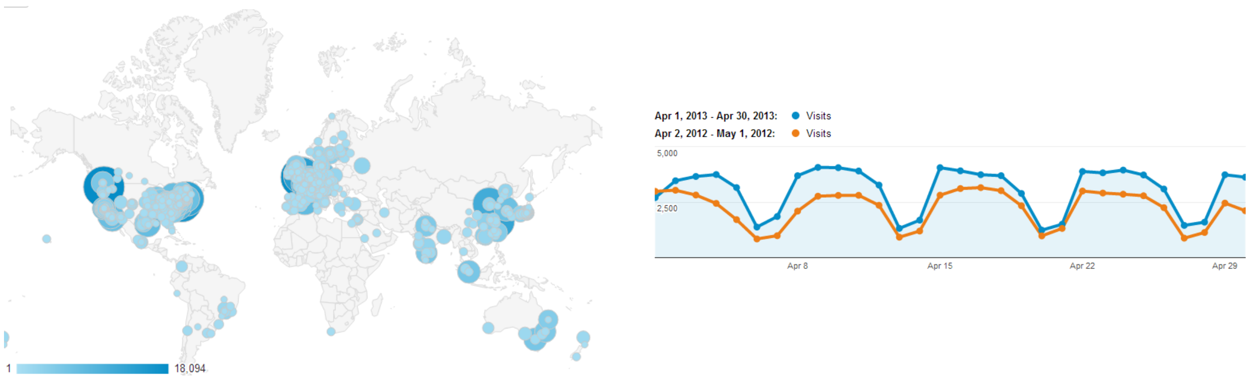


Figure 2.1: *Bioconductor* Google analytics, 1-month access, 10 December 2012. Left: access by country. Right: daily access in 2011 (orange) and 2012 (blue).

Table 2.1: Selected *Bioconductor* packages for high-throughput sequence analysis.

Concept	Packages
Data representation	*IRanges*, *GenomicRanges*, *Biostrings*, *BSgenome*, *VariantAnnotation*.
Input / output	*ShortRead* (FASTQ), *Rsamtools* (BAM), *rtracklayer* (GFF, WIG, BED), *VariantAnnotation* (VCF).
Annotation	*AnnotationHub*, *biomaRt*, *GenomicFeatures*, *TxDb.**, *org.**, *ChIPpeakAnno*, *VariantAnnotation*.
Alignment	*gmapR*, *Rsubread*, *Biostrings*.
Visualization	*ggbio*, *Gviz*.
Quality assessment	*qrqc*, *seqbias*, *ReQON*, *htSeqTools*, *TEQC*, *ShortRead*.
RNA-seq	*BitSeq*, *cqn*, *cummeRbund*, *DESeq2*, *DEXSeq*, *EDASeq*, *edgeR*, *gage*, *goseq*, *iASeq*, *tweeDEseq*.
ChIP-seq, etc.	*BayesPeak*, *baySeq*, *ChIPpeakAnno*, *chipseq*, *ChIPseqR*, *ChIPsim*, *CSAR*, *DiffBind*, *MEDIPS*, *mosaics*, *NarrowPeaks*, *nucleR*, *PICS*, *PING*, *REDseq*, *Repitools*, *TSSi*.
Variants	*VariantAnnotation*, *VariantTools*, *gmapR*
SNPs	*snpStats*, *GWASTools*, *SeqVarTools*, *hapFabia*, *GGtools*
Copy number	*cn.mops*, *genoset*, *fastseq*, *CNAnorm*, *exomeCopy*, *seqmentSeq*.
Motifs	*MotifDb*, *BCRANK*, *cosmo*, *MotIV*, *seqLogo*, *rGADEM*.
3C, etc.	*HiTC*, *r3Cseq*.
Microbiome	*phyloseq*, *DirichletMultinomial*, *clstutils*, *manta*, *mcaGUI*.
Work flows	*QuasR*, *easyRNASeq*, *ArrayExpressHTS*, *oneChannelGUI*.
Database	*SRAdb*, *GEOquery*.

stallation instructions), `http://bioconductor.org/packages/release/bioc/` (current software packages), `http://bioconductor.org/help/mailing-list/` (mailing lists).

2.1 Bioconductor for high-throughput sequence analysis

Recent technological developments introduce high-throughput sequencing approaches. A variety of experimental protocols and analysis work flows address gene expression, regulation, and encoding of genetic variants. Experimental protocols produce a large number (tens to hundreds of millions per sample) of short (e.g., 35-150, single or paired-end) nucleotide sequences. These are aligned to a reference or other genome. Analysis work flows use the alignments to infer levels of gene expression (RNA-seq), binding of regulatory elements to genomic locations (ChIP-seq), or prevalence of structural variants (e.g., SNPs, short indels, large-scale genomic rearrangements). Sample sizes range from minimal replication (e.g,. 2 samples per treatment group) to thousands of individuals.

Table 2.1 enumerates many of the packages available for sequence analysis. The table includes packages for representing sequence-related data (e.g., *GenomicRanges*, *Biostrings*), as well as domain-specific analysis such as RNA-seq (e.g., *edgeR*, *DEXSeq*), ChIP-seq (e.g,. *ChIPpeakAnno*, *DiffBind*), variants (e.g., *VariantAnnotation*, *VariantTools*), and SNPs and copy number variation (e.g., *genoset*, *ggtools*).

S4 Classes and methods *Bioconductor* makes extensive use of 'S4' classes. Essential operations are sketched in Table 2.2. *Bioconductor* has tried to develop and use 'best practices' for S4 classes. Usually instances are created by a call to a *constructor*, such as `GRanges` (an object representing genomic ranges, with information on sequence, strand, start, and end coordinate of each range), or are returned by a function call that makes the object 'behind the scenes' (e.g., `readFastq`). Objects can have complicated structure, but users are not expected to have to concern themselves with the internal representation, just as the details of the S3 object returned by the `lm` function are not of direct concern. Instead, one might query the object to retrieve information; functions providing this functionality are sometimes called *accessors*, e.g., `seqnames`; the data that is returned by the accessor may involve some calculation, e.g., querying a data base, that the user can remain blissfully unaware of. It can be important to appreciate that an object can be related to other objects, in particular inheriting parts of its internal structure and external behavior from other classes. For

Table 2.2: Using S4 classes and methods.

Best practices	
`gr <- GRanges()`	'Constructor'; create an instance of the *GRanges* class
`seqnames(gr)`	'Accessor', extract information from an instance
`countOverlaps(gr1, gr2)`	A *method* implementing a *generic* with useful functionality
Older packages	
`s <- new("MutliSet)`	A constructor
`s@annotation`	A 'slot' accessor
Help	
`class(gr)`	Discover class of instance
`getClass(gr)`	Display class structure, e.g., inheritance
`showMethods(findOverlaps)`	Classes for which methods of `findOverlaps` are implemented
`showMethods(class="GRanges", where=search())`	Generics with methods implemented for the *GRanges* class, limited to currently loaded packages.
`class?GRanges`	Documentation for the *GRanges* class.
`method?"findOverlaps,GRanges,GRanges"`	Documentation for the `findOverlaps` method when the two arguments are both *GRanges* instances.
`selectMethod(findOverlaps, c("GRanges", "GRanges"))`	View source code for the method, including method 'dispatch'

instance, the *GRanges* class inherits structure and behavior from the *GenomicRanges* and *IRanges* classes. The details of structural inheritance should not be important to the user, but the fact that once class inherits from another can be useful information to know especially when navigating the help system.

One often calls a function in which one or more objects are arguments, e.g., `countOverlaps` can take two *GRanges* instances. The role of the function is to transform inputs into outputs. In the case of `countOverlaps` the transformation is to summarize the number of ranges in the second argument (the `subject` function argument) overlap with ranges in the first argument (the `query`) argument. This establishes a kind of contract, e.g., the return value of `countOverlaps` should be a non-negative integer vector, with as many elements as there are ranges in the `query` argument, and with a one-to-one correspondence between elements in the `query` input argument and the output. Having established such a contract, it can be convenient to write variations of `countOverlaps` that fulfill the contract but for different objects, e.g., when the arguments are instances of class *IRanges*, which do not have information about chromosomal sequence or strand. To indicate that the same contract is being fulfilled, and perhaps to simplify software development, one typically makes `countOverlaps` a *generic* function, and implements *methods* for different types of arguments.

Attending courses and reading vignette pages are obviously an excellent way to get an initial orientation about available classes and methods. It can be very helpful, as one becomes more proficient, to use the interactive help system to discover what can be done with the objects one has or the functions one knows about. The `showMethods` function is a key entry point into discovery of available methods, e.g., `showMethods("countOverlaps")` to show methods defined on the `countOverlaps` generic, or `showMethods(class="GRanges", where=search())` to discover methods available to transform *GRanges* instances. The definition of a method can be retrieved as

```
> selectMethod(countOverlaps, c("GRanges", "GRanges"))
```

Exercise 4

Load the GenomicRanges package.

a. *Use `getClass` to discover the class structure of GRanges, paying particular attention to inheritance relationships summarized in the "Extends:" section of the display.*
b. *Use `showMethods` to see what methods are defined for the `countOverlaps` function.*
c. *There are many methods defined for `countOverlaps`, but none are listed for the GRanges,GRanges combination of arguments. Yet `countOverlaps`* does *work when provided with two GRanges arguments. Why is that?*

Solution: Here we load the package and ask about class structure.

```
> library(GenomicRanges)
> getClass("GRanges")

Class "GRanges" [package "GenomicRanges"]

Slots:

Name:         seqnames          ranges          strand elementMetadata
Class:             Rle         IRanges             Rle       DataFrame

Name:          seqinfo        metadata
Class:         Seqinfo            list

Extends:
Class "GenomicRanges", directly
Class "Vector", by class "GenomicRanges", distance 2
Class "GenomicRangesORmissing", by class "GenomicRanges", distance 2
Class "GenomicRangesORGRangesList", by class "GenomicRanges", distance 2
Class "Annotated", by class "GenomicRanges", distance 3
```

GRanges extends several classes, including *GenomicRanges*. Methods defined on the countOverlaps generic can be discovered with

```
> showMethods("countOverlaps")

Function: countOverlaps (package IRanges)
query="ANY", subject="Vector"
query="ANY", subject="missing"
query="GAlignmentPairs", subject="GAlignmentPairs"
query="GAlignmentPairs", subject="Vector"
query="GAlignments", subject="GAlignments"
query="GAlignments", subject="GRangesList"
query="GAlignments", subject="GenomicRanges"
query="GAlignments", subject="Vector"
query="GAlignmentsList", subject="GAlignmentsList"
query="GAlignmentsList", subject="Vector"
query="GRanges", subject="GRangesList"
query="GRangesList", subject="GAlignments"
query="GRangesList", subject="GRanges"
query="GRangesList", subject="GRangesList"
query="GRangesList", subject="Vector"
query="GenomicRanges", subject="GAlignments"
query="GenomicRanges", subject="GenomicRanges"
query="GenomicRanges", subject="Vector"
query="RangedData", subject="RangedData"
query="RangedData", subject="RangesList"
query="RangesList", subject="IntervalForest"
query="RangesList", subject="RangedData"
query="RangesList", subject="RangesList"
query="SummarizedExperiment", subject="SummarizedExperiment"
query="SummarizedExperiment", subject="Vector"
query="Vector", subject="GAlignmentPairs"
query="Vector", subject="GAlignments"
query="Vector", subject="GAlignmentsList"
query="Vector", subject="GRangesList"
```

```
query="Vector", subject="GenomicRanges"
query="Vector", subject="SummarizedExperiment"
query="Vector", subject="ViewsList"
query="ViewsList", subject="Vector"
query="ViewsList", subject="ViewsList"
```

Note that there is no method defined for the *GRanges,GRanges* combination of arguments. Yet `countOverlaps` *does* work. . . (skim over the details of these objects; we are using a constructor to make genomic ranges on plus strand of chromosome 1; there are two ranges in `gr0`, and one in `gr1`.

```
> gr0 <- GRanges("chr1", IRanges(start=c(10, 20), width = 5), "+")
> gr1 <- GRanges("chr1", IRanges(start=12, end=18), "+")
> countOverlaps(gr0, gr1)
```

```
[1] 1 0
```

`gr1` overlaps the first range of `gr0`, but not the second, and we end up with a vector of counts `c(1, 0)`. The reason that this 'works' is because of inheritance – *GRanges* extends *GenomicRanges*, and we end up selecting the inherited method *countOverlaps,GenomicRanges,GenomicRanges-method*.

Exercise 5
To illustrate how help work with S4 classes and generics, consider the DNAStringSet class `complement` *generic in the Biostrings package:*

```
> library(Biostrings)
> showMethods(complement)
```

```
Function: complement (package Biostrings)
x="DNAString"
x="DNAStringSet"
x="MaskedDNAString"
x="MaskedRNAString"
x="RNAString"
x="RNAStringSet"
x="XStringViews"
```

(Most) methods defined on the `DNAStringSet` *class of Biostrings and available on the current search path can be found with ■S4-showMethods, eval=FALSE■= showMethods(class=”DNAStringSet”, where=search()) Obtaining help on S4 classes and methods requires syntax such as*

```
> class ? DNAStringSet
> method ? "complement,DNAStringSet"
```

The specification of method and class in the latter must not contain a space after the comma.

2.2 Sequencing technologies and work flows

Recent technological developments introduce high-throughput sequencing approaches. A variety of experimental protocols and analysis work flows address gene expression, regulation, and encoding of genetic variants. Experimental protocols produce a large number (tens of millions per sample) of short (e.g., 35-150, single or paired-end) nucleotide sequences. These are aligned to a reference or other genome. Analysis work flows use the alignments to infer levels of gene expression (RNA-seq), binding of regulatory elements to genomic locations (ChIP-seq), or prevalence of structural variants (e.g., SNPs, short indels, large-scale genomic rearrangements). Sample sizes range from minimal replication (e.g,. 2 samples per treatment group) to thousands of individuals.

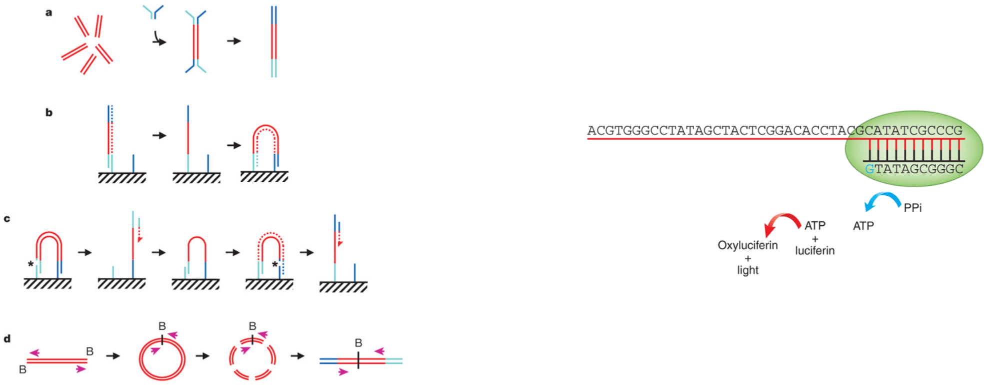


Figure 2.2: High-throughput sequencing. Left: Illumina bridge PCR [3]; mis-call errors. Right: Roche 454 [33]; homopolymer errors.

2.2.1 Technologies

The most common 'second generation' technologies readily available to labs are

- Illumina single- and paired-end reads. Short (≈ 100 per end) and very numerous. Flow cell, lane, bar-code.
- Roche 454. 100's of nucleotides, 100,000's of reads.
- Life Technologies SOLiD. Unique 'color space' model.
- Complete Genomics. Whole genome sequence / variants / etc as a service; end user gets derived results.

Figure 2.2 illustrates Illumina and 454 sequencing. *Bioconductor* has good support for Illumina and some support for Roche 454 sequencing products, and very good support for derived data such as aligned reads or called variants; use of SOLiD color space reads in *Bioconductor* requires conversion to FASTQ files that undermine the benefit of the color space model.

All second-generation technologies rely on PCR and other techniques to generate reads from samples that represent aggregations of many DNA molecules. 'Third-generation' technologies shift to single-molecule sequencing, with relevant players including Pacific Biosciences and IonTorent. This data is not widely available, and will not be discussed further.

The most common data in *Bioconductor* work flows is from Illumina sequencers. Reads are either single-end or paired-end. Single-end reads represent $30 - 130$ nucleotides sequenced from DNA that has been sheared into ~ 300 nucleotide fragments. Paired-end reads represent $30 - 130$ nucleotide reads that are paired, and from both ends of the 300 nucleotide fragment.

Sequence data can be derived from a tremendous diversity of experiments. Some of the most common include:

RNA-seq Sequencing of reverse-complemented mRNA from the entire expressed transcriptome, typically. Used for *differential expression* studies like micro-arrays, or for *novel transcript discovery*.

DNA-seq Sequencing of whole or targeted (e.g., exome) genomic DNA. Common goals include *SNP* detection, *indel* and other structural polymorphisms, and *CNV* (copy number variation). DNA-seq is also used for *de novo* assembly, but *de novo* assembly is not an area where *Bioconductor* contributes.

ChIP-seq ChIP (chromatin immuno-precipitation) is used to enrich genomic DNA for regulatory elements, followed by sequencing and mapping of the enriched DNA to a reference genome. The initial statistical challenge is to identify regions where the mapped reads are enriched relative to a sample that did not undergo ChIP[29]; a subsequent task is to identify differential binding across a designed experiment, e.g., [32].

Metagenomics Metagenomic sequencing generates sequences from samples containing multiple species, typically microbial communities sampled from niches such as the human oral cavity. Goals include inference of *species composition* (when sequencing typically targets phylogenetically informative genes such as 16S) or *metabolic contribution*.

2.2.2 Data and work flows

At a very high level, a typical work flow starts with an sapmles from a designed experiment being sequences to produce FASTA files of raw reads and their quality scores. These are aligned to a reference genome, with the alignments represented in BAM files. For DNA-seq experiments, BAM files may be further summarized to called variants represented in VCF files. For RNA-seq BAM files and VCF files, One typically interprets reads in BAM files or called variants in VCF files

Table 2.3: Common file types (e.g., `http://genome.ucsc.edu/FAQ/FAQformat.html`) and *Bioconductor* packages used for input.

File	Description	Package
FASTQ	Unaligned sequences: identifier, sequence, and encoded quality score tuples	*ShortRead*
BAM	Aligned sequences: identifier, sequence, reference sequence name, strand position, cigar and additional tags	*Rsamtools*
GFF, GTF	Gene annotations: reference sequence name, data source, feature type, start and end positions, strand, etc.	*rtracklayer*
BED	Range-based annotation: reference sequence name, start, end coordinates.	*rtracklayer*
WIG, bigWig	'Continuous' single-nucleotide annotation.	*rtracklayer*
VCF	Called single nucleotide, indel, copy number, and structural variants, often compressed and indexed (with *Rsamtools* `bgzip`, `indexTabix`)	*VariantAnnotation*

in terms of known whole-genome annotations, these are represented by GFF, GTF, BED, WIG, or bigWig files. *R* and *Bioconductor* are useful for accessing each of these data types, as summarized in Table 2.3.

Many relevant statistical computations involve extracting summaries of data from BAM (e.g., reads overlapping regions of interest) or VCF (e.g., variant frequencies across individuals in large population samples) files. Important statistical issues arising during analysis, (e.g., assessment of quality or technological bias) require access to details of this data.

Common analyses often use well-established third-party tools for initial stages of the analysis; some of these have *Bioconductor* counterparts that are particularly useful when the question under investigation does not meet the assumptions of other facilities. Some common work flows (a more comprehensive list is available on the SeqAnswers wiki[1]) include:

ChIP-seq ChIP-seq experiments typically use DNA sequencing to identify regions of genomic DNA enriched in prepared samples relative to controls. A central task is thus to identify peaks, with common tools including *MACS* and *PeakRanger*.

RNA-seq In addition to the aligners mentioned above, RNA-seq for differential expression might use the *HTSeq*[2] python tools for counting reads within regions of interest (e.g., known genes) or a pipeline such as the *bowtie* (basic alignment) / *tophat* (splice junction mapper) / *cufflinks* (esimated isoform abundance) (e.g., [3]) or *RSEM*[4] suite of tools for estimating transcript abundance.

DNA-seq especially variant calling can be facilitated by software such as the GATK[5] toolkit.

There are many *R* packages that replace or augment the analyses outlined above, as summarized in Table 2.1.

Programs such as those outlined the previous paragraph often rely on information about gene or other structure as input, or produce information about chromosomal locations of interesting features. The GTF and BED file formats are common representations of this information. Representing these files as *R* data structures is often facilitated by the *rtracklayer* package. Variants are very commonly represented in VCF (Variant Call Format) files; these are explored in other *Bioconductor* tutorials.

2.3 Strings and reads

2.3.1 DNA (and other) Strings with the Biostrings package

The *Biostrings* package provides tools for working with sequences. The essential data structures are *DNAString* and *DNAStringSet*. The *Biostrings* package contains additional classes for representing amino acid and general biological strings. The *BSgenome* and related packages (e.g., *BSgenome.Dmelanogaster.UCSC.dm3*) are used to represent whole-genome sequences. Table 2.5 summarizes common operations; the following exercise explores these packages.

[1] `http://seqanswers.com/wiki/RNA-Seq`

[2] `http://www-huber.embl.de/users/anders/HTSeq/doc/overview.html`

[3] `http://bowtie-bio.sourceforge.net/index.shtml`

[4] `http://deweylab.biostat.wisc.edu/rsem/`

[5] `http://www.broadinstitute.org/gatk/`

Table 2.4: Selected *Bioconductor* packages for representing strings and reads.

Package	Description
Biostrings	Classes (e.g., *DNAStringSet*) and methods (e.g., `alphabetFrequency`, `pairwiseAlignment`) for representing and manipulating DNA and other biological sequences.
BSgenome	Representation and manipulation of large (e.g., whole-genome) sequences.
ShortRead	I/O and manipulation of FASTQ files.

Table 2.5: Operations on strings in the *Biostrings* package.

	Function	Description
Access	`length`, `names`	Number and names of sequences
	`[`, `head`, `tail`, `rev`	Subset, first, last, or reverse sequences
	`c`	Concatenate two or more objects
	`width`, `nchar`	Number of letters of each sequence
	`Views`	Light-weight sub-sequences of a sequence
Compare	`==`, `!=`, `match`, `%in%`	Element-wise comparison
	`duplicated`, `unique`	Analog to `duplicated` and `unique` on character vectors
	`sort`, `order`	Locale-independent sort, order
	`split`, `relist`	Split or relist objects to, e.g., *DNAStringSetList*
Edit	`subseq`, `subseq<-`	Extract or replace sub-sequences in a set of sequences

	`reverse`, `complement`
	`reverseComplement`
		Reverse, complement, or reverse-complement DNA

	`translate`	Translate DNA to Amino Acid sequences
	`chartr`	Translate between letters
	`replaceLetterAt`	Replace letters at a set of positions by new letters
	`trimLRPatterns`	Trim or find flanking patterns

Count	`alphabetFrequency`
	`letterFrequency`
	`letterFrequencyInSlidingView`
		Tabulate letter occurrence

	`consensusMatrix`	Nucleotide × position summary of letter counts

	`dinucleotideFrequency`
	`trinucleotideFrequency`
	`oligonucleotideFrequency`
		2-mer, 3-mer, and k-mer counting

	`nucleotideFrequencyAt`	Nucleotide counts at fixed sequence positions

Match	`matchPattern`, `countPattern`
	`vmatchPattern`, `vcountPattern`
		Short patterns in one or many (`v*`) sequences

	`matchPDict`, `countPDict`
	`whichPDict`, `vcountPDict`
	`vwhichPDict`
		Short patterns in one or many (`v*`) sequences (mismatch only)

	`pairwiseAlignment`	Needleman-Wunsch, Smith-Waterman, etc. pairwise alignment
	`matchPWM`, `countPWM`	Occurrences of a position weight matrix
	`matchProbePair`	Find left or right flanking patterns
	`findPalindromes`	Palindromic regions in a sequence. Also `findComplementedPalindromes`
	`stringDist`	Levenshtein, Hamming, or pairwise alignment scores
I/O	`readDNAStringSet` `writeXStringSet`	FASTA (or sequence only from FASTQ). Also `readBStringSet`, `readRNAStringSet`, `readAAStringSet`
	`writePairwiseAlignments`	Write `pairwiseAlignment` as "pair" format
	`readDNAMultipleAlignment` `write.phylip`	Multiple alignments (FASTA, "stockholm", or "clustal"). Also `readRNAMultipleAlignment`, `readAAMultipleAlignment`

Exercise 6

The objective of this exercise is to calculate the GC content of the exons of a single gene. We jump into the middle of some of the data structures common in Bioconductor; these are introduced more thoroughly in later exercises..

Load the BSgenome.Dmelanogaster.UCSC.dm3 data package, containing the UCSC representation of D. melanogaster *genome assembly dm3. Discover the content of the package by evaluating `Dmelanogaster`.*

Load the SequenceAnalysisData package, and evaluate the command `data(ex)` to load an example of a GRangesList object. the GRangesList represents coordinates of exons in the D. melanogaster *genome, grouped by gene.*

Look at `ex[1]`. These are the genomic coordinates of the first gene in the `ex` object. Load the D. melanogaster *chromosome that this gene is on by subsetting the `Dmelanogaster` object.*

Use `Views` to create views on to the chromosome that span the start and end coordinates of all exons in the first gene; the start and end coordinates are accessed with `start(ex[[1]])` and similar.

Develop a function `gcFunction` to calculate GC content. Use this to calculate the GC content in each of the exons.

Solution: Here we load the *D. melanogaster* genome, select a single chromosome, and create `Views` that reflect the ranges of the FBgn0002183.

```
> library(BSgenome.Dmelanogaster.UCSC.dm3)
> Dmelanogaster

Fly genome
|
| organism: Drosophila melanogaster (Fly)
| provider: UCSC
| provider version: dm3
| release date: Apr. 2006
| release name: BDGP Release 5
|
| single sequences (see '?seqnames'):
|   chr2L      chr2R      chr3L      chr3R      chr4       chrX       chrU
|   chrM       chr2LHet   chr2RHet   chr3LHet   chr3RHet   chrXHet    chrYHet
|   chrUextra
|
| multiple sequences (see '?mseqnames'):
|   upstream1000  upstream2000  upstream5000
|
| (use the '$' or '[[' operator to access a given sequence)

> library(SequenceAnalysisData)
> data(ex)
> ex[1]

GRangesList of length 1:
$FBgn0002183
GRanges with 9 ranges and 0 metadata columns:
      seqnames             ranges strand
         <Rle>          <IRanges>  <Rle>
  [1]    chr3L [1871574, 1871917]      -
  [2]    chr3L [1872354, 1872470]      -
  [3]    chr3L [1872582, 1872735]      -
  [4]    chr3L [1872800, 1873062]      -
  [5]    chr3L [1873117, 1873983]      -
  [6]    chr3L [1874041, 1875218]      -
  [7]    chr3L [1875287, 1875586]      -
  [8]    chr3L [1875652, 1875915]      -
  [9]    chr3L [1876110, 1876336]      -
```

```
---
seqlengths:
     chr2L   chr2LHet     chr2R  chr2RHet ...    chrXHet   chrYHet      chrM
  23011544     368872  21146708   3288761 ...     204112    347038     19517
```

```
> nm <- "chr3L"
> chr <- Dmelanogaster[[nm]]
> v <- Views(chr, start=start(ex[[1]]), end=end(ex[[1]]))
```

Here is the gcFunction helper function to calculate GC content:

```
> gcFunction <-
+     function(x)
+ {
+     alf <- alphabetFrequency(x, as.prob=TRUE)
+     rowSums(alf[,c("G", "C")])
+ }
```

The gcFunction is really straight-forward: it invokes the function alphabetFrequency from the *Biostrings* package. This returns a simple matrix of exon × nucleotide probabilities. The row sums of the G and C columns of this matrix are the GC contents of each exon.

The subject GC content is

```
> subjectGC <- gcFunction(v)
```

2.3.2 Reads and the ShortRead package

Short read formats The Illumina GAII and HiSeq technologies generate sequences by measuring incorporation of florescent nucleotides over successive PCR cycles. These sequencers produce output in a variety of formats, but *FASTQ* is ubiquitous. Each read is represented by a record of four components:

```
@SRR031724.1 HWI-EAS299_4_30M2BAAXX:5:1:1513:1024 length=37
GTTTTGTCCAAGTTCTGGTAGCTGAATCCTGGGGCGC
+SRR031724.1 HWI-EAS299_4_30M2BAAXX:5:1:1513:1024 length=37
IIIIIIIIIIIIIIIIIIIIIIIIIIII+HIIII<IE
```

The first and third lines (beginning with @ and + respectively) are unique identifiers. The second and fourth lines are the nucleotides and qualities of each cycle in the read. This information is given in 5' to 3' orientation as seen by the sequencer. A letter N in the sequence is used to signify bases that the sequencer was not able to call. The fourth line of the FASTQ record encodes the quality (confidence) of the corresponding base call. The quality score is encoded following one of several conventions, with the general notion being that letters later in the visible ASCII alphabet

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

are of higher quality; this is developed further below. Both the sequence and quality scores may span multiple lines.

Short reads in R The *ShortRead* package provides tools for working with short reads. Data can be input using readFastq or FastqSampler and FastqStreamer; the later are preferred because they allow input of subsets of these large files. Functions for accessing the read and quality scores are provided, as are other methods to perform common operations (Tabl 2.6. Reads and quality scores are represented by *DNAStringSet* and related objects, so the methods in Table 2.6 are available.

Exercise 7
Use the file path bigdata *and the* file.path *and* dir *functions to locate the fastq file from [4] (the file was obtained as described in the pasilla experiment data package).*

Table 2.6: Operations on FASTQ (and FASTA) files from the *ShortRead* and `Rsamtools` packages (see also Table 2.5).

	Function	Description
Classes	*FaFile*	Reference a FASTA flie
	FastqFile	Reference a FASTQ file
	ShortReadQ	Read and quality scores
I/O	`readFastq`, `writeFastq`	Read and write fastq files
	`FastqSampler`, `FastqStreamer`	Sample from or iterate through a large file
	`scanFa`	Selectively input an (indexed) FASTA file
	`razip`	Compress a FASTA file for random access
Access	`sread`, `quality`, `id`	Reads, quality scores, and identifiers
Calculation	`alphabetByCycle`	Nucleotide frequency by cycle
	`alphabetScore`	Log read score across cycle
	`tables`	Tabulate read occurrence
Quality	`SRFilter`	Filter reads based on diverse criteria
	`trimEnds`, `trimTails`, `trimTailw`	Trim reads based on quality.
	`qa`, `report`	Prepare a quality report

Input the fastq files using `readFastq` from the ShortRead package.

Use `alphabetFrequency` to summarize the GC content of all reads (hint: use the `sread` accessor to extract the reads, and the `collapse=TRUE` argument to the `alphabetFrequency` function). Using the helper function `gcFunction` defined elsewhere in this document, draw a histogram of the distribution of GC frequencies across reads.

Use `alphabetByCycle` to summarize the frequency of each nucleotide, at each cycle. Plot the results using `matplot`, from the graphics package.

As an advanced exercise, and if on Mac or Linux, use the parallel package and `mclapply` to read and summarize the GC content of reads in two files in parallel.

Solution: Discovery:

```
> library(ShortRead)
> bigdata <- system.file("bigdata", package="SequenceAnalysisData")
> dir(bigdata)
```

```
[1] "bam"                      "dm3.ensGene.txdb.sqlite"
[3] "fastq"
```

```
> fls <- dir(file.path(bigdata, "fastq"), full=TRUE)
```

Input:

```
> fq <- readFastq(fls[1])
```

GC content:

```
> alf0 <- alphabetFrequency(sread(fq), as.prob=TRUE, collapse=TRUE)
> sum(alf0[c("G", "C")])
```

```
[1] 0.5457237
```

A histogram of the GC content of individual reads is obtained with

```
> gc <- gcFunction(sread(fq))
> hist(gc)
```

Alphabet by cycle:

```
> abc <- alphabetByCycle(sread(fq))
> matplot(t(abc[c("A", "C", "G", "T"),]), type="l")
```

Advanced (Mac, Linux only): processing on multiple cores.

```
> library(parallel)
> gc0 <- mclapply(fls, function(fl) {
+   fq <- readFastq(fl)
+   gc <- gcFunction(sread(fq))
+   table(cut(gc, seq(0, 1, .05)))
+ })
> ## simplify list of length 2 to 2-D array
> gc <- simplify2array(gc0)
> matplot(gc, type="s")
```

Exercise 8
Use `quality` *to extract the quality scores of the short reads. Interpret the encoding qualitatively.*

Convert the quality scores to a numeric matrix, using `as`*. Inspect the numeric matrix (e.g., using* `dim`*) and understand what it represents.*

Use `colMeans` *to summarize the average quality score by cycle. Use* `plot` *to visualize this.*

Solution:

```
> head(quality(fq))

class: FastqQuality
quality:
  A BStringSet instance of length 6
    width seq
[1]    37 IIIIIIIIIIIIIIIIIIIIIIIIIIII+HIIII<IE
[2]    37 IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII
[3]    37 IIIIIIIIIIIIIIIIIIIIII'IIIIIGBIIII2I+
[4]    37 IIIIIIIIIIIIIIIIIIIIIIII,II*E,&4HI++B
[5]    37 IIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIIII&.$
[6]    37 III.IIIIIIIIIIIIIIIIIII%IIE(-EIH<IIII

> qual <- as(quality(fq), "matrix")
> dim(qual)

[1] 1000000      37

> plot(colMeans(qual), type="b")
```

Exercise 9
As an independent exercise, visit the qrqc landing page and explore the package vignette. Use the qrqc package (you may need to install this) to generate base and average quality plots for the data, like those in the report generated by ShortRead.

2.4 Ranges and alignments

Ranges describe both features of interest (e.g., genes, exons, promoters) and reads aligned to the genome. *Bioconductor* has very powerful facilities for working with ranges, some of which are summarized in Table 2.7.

Table 2.7: Selected *Bioconductor* packages for representing and manipulating ranges.

Package	Description
IRanges	Defines important classes (e.g., *IRanges*, *Rle*) and methods (e.g., `findOverlaps`, `countOverlaps`) for representing and manipulating ranges of consecutive values. Also introduces *DataFrame*, *SimpleList* and other classes tailored to representing very large data.
GenomicRanges	Range-based classes tailored to sequence representation (e.g., *GRanges*, *GRangesList*), with information about strand and sequence name.
GenomicFeatures	Foundation for manipulating data bases of genomic ranges, e.g., representing coordinates and organization of exons and transcripts of known genes.

2.4.1 Ranges and the GenomicRanges package

Next-generation sequencing data consists of a large number of short reads. These are, typically, aligned to a reference genome. Basic operations are performed on the alignment, asking e.g., how many reads are aligned in a genomic range defined by nucleotide coordinates (e.g., in the exons of a gene), or how many nucleotides from all the aligned reads cover a set of genomic coordinates. How is this type of data, the aligned reads and the reference genome, to be represented in *R* in a way that allows for effective computation?

The *IRanges*, *GenomicRanges*, and *GenomicFeatures Bioconductor* packages provide the essential infrastructure for these operations; we start with the *GRanges* class, defined in *GenomicRanges*.

GRanges Instances of *GRanges* are used to specify genomic coordinates. Suppose we wish to represent two *D. melanogaster* genes. The first is located on the positive strand of chromosome 3R, from position 19967117 to 19973212. The second is on the minus strand of the X chromosome, with 'left-most' base at 18962306, and right-most base at 18962925. The coordinates are *1-based* (i.e., the first nucleotide on a chromosome is numbered 1, rather than 0), *left-most* (i.e., reads on the minus strand are defined to 'start' at the left-most coordinate, rather than the 5' coordinate), and *closed* (the start and end coordinates are included in the range; a range with identical start and end coordinates has width 1, a 0-width range is represented by the special construct where the end coordinate is one less than the start coordinate).

A complete definition of these genes as *GRanges* is:

```
> genes <- GRanges(seqnames=c("3R", "X"),
+                  ranges=IRanges(
+                      start=c(19967117, 18962306),
+                      end=c(19973212, 18962925)),
+                  strand=c("+", "-"),
+                  seqlengths=c(`3R`=27905053L, `X`=22422827L))
```

The components of a *GRanges* object are defined as vectors, e.g., of `seqnames`, much as one would define a *data.frame*. The start and end coordinates are grouped into an *IRanges* instance. The optional `seqlengths` argument specifies the maximum size of each sequence, in this case the lengths of chromosomes 3R and X in the 'dm2' build of the *D. melanogaster* genome. This data is displayed as

```
> genes
```

```
GRanges with 2 ranges and 0 metadata columns:
      seqnames                 ranges strand
         <Rle>              <IRanges>  <Rle>
  [1]       3R [19967117, 19973212]      +
  [2]        X [18962306, 18962925]      -
  ---
  seqlengths:
         3R        X
   27905053 22422827
```

For the curious, the gene coordinates and sequence lengths are derived from the *org.Dm.eg.db* package for genes with Flybase identifiers FBgn0039155 and FBgn0085359, using the annotation facilities described in Chapter 6.1.

The *GRanges* class has many useful methods defined on it. Consult the help page

```
> ?GRanges
```

and package vignettes (especially 'An Introduction to *GenomicRanges*')

```
> vignette(package="GenomicRanges")
```

for a comprehensive introduction. A *GRanges* instance can be subset, with accessors for getting and updating information.

```
> genes[2]

GRanges with 1 range and 0 metadata columns:
      seqnames               ranges strand
         <Rle>            <IRanges>  <Rle>
  [1]        X [18962306, 18962925]      -
  ---
  seqlengths:
         3R        X
   27905053 22422827

> strand(genes)

factor-Rle of length 2 with 2 runs
  Lengths: 1 1
  Values : + -
Levels(3): + - *

> width(genes)

[1] 6096  620

> length(genes)

[1] 2

> names(genes) <- c("FBgn0039155", "FBgn0085359")
> genes  # now with names

GRanges with 2 ranges and 0 metadata columns:
              seqnames               ranges strand
                 <Rle>            <IRanges>  <Rle>
  FBgn0039155       3R [19967117, 19973212]      +
  FBgn0085359        X [18962306, 18962925]      -
  ---
  seqlengths:
         3R        X
   27905053 22422827
```

`strand` returns the strand information in a compact representation called a *run-length encoding*, this is introduced in greater detail below. The 'names' could have been specified when the instance was constructed; once named, the *GRanges* instance can be subset by name like a regular vector.

As the `GRanges` function suggests, the *GRanges* class extends the *IRanges* class by adding information about `seqnames`, `strand`, and other information particularly relevant to representing ranges that are on genomes. The *IRanges* class and related data structures (e.g., *RangedData*) are meant as a more general description of ranges defined in an arbitrary space. Many methods implemented on the *GRanges* class are 'aware' of the consequences of genomic location, for instance treating ranges on the minus strand differently (reflecting the 5' orientation imposed by DNA) from ranges on the plus strand.

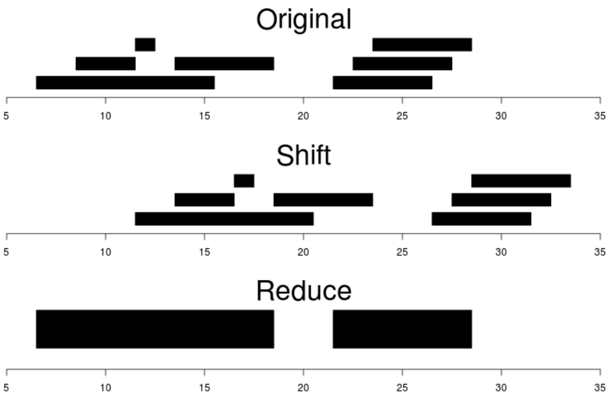


Figure 2.3: Ranges

Operations on ranges The *GRanges* class has many useful methods from the *IRanges* class; some of these methods are illustrated here. We use *IRanges* to illustrate these operations to avoid complexities associated with strand and seqnames, but the operations are comparable on *GRanges*. We begin with a simple set of ranges:

```
> ir <- IRanges(start=c(7, 9, 12, 14, 22:24),
+               end=c(15, 11, 12, 18, 26, 27, 28))
```

These and some common operations are illustrated in the upper panel of Figure 2.3 and summarized in Table 2.8.

Methods on ranges can be grouped as follows:

Intra-range methods act on each range independently. These include `flank`, `narrow`, `reflect`, `resize`, `restrict`, and `shift`, among others. An illustration is `shift`, which translates each range by the amount specified by the `shift` argument. Positive values shift to the right, negative to the left; `shift` can be a vector, with each element of the vector shifting the corresponding element of the *IRanges* instance. Here we shift all ranges to the right by 5, with the result illustrated in the middle panel of Figure 2.3.

```
> shift(ir, 5)
```

```
IRanges of length 7
    start end width
[1]    12  20     9
[2]    14  16     3
[3]    17  17     1
[4]    19  23     5
[5]    27  31     5
[6]    28  32     5
[7]    29  33     5
```

Inter-range methods act on the collection of ranges as a whole. These include `disjoin`, `reduce`, `gaps`, and `range`. An illustration is `reduce`, which reduces overlapping ranges into a single range, as illustrated in the lower panel of Figure 2.3.

```
> reduce(ir)
```

```
IRanges of length 2
    start end width
[1]     7  18    12
[2]    22  28     7
```

coverage is an inter-range operation that calculates how many ranges overlap individual positions. Rather than returning ranges, `coverage` returns a compressed representation (run-length encoding)

```
> coverage(ir)
```

Table 2.8: Common operations on *IRanges*, *GRanges* and *GRangesList*.

Category	Function	Description
Accessors	`start`, `end`, `width`	Get or set the starts, ends and widths
	`names`	Get or set the names
	`mcols`, `metadata`	Get or set metadata on elements or object
	`length`	Number of ranges in the vector
	`range`	Range formed from min start and max end
Ordering	`<`, `<=`, `>`, `>=`, `==`, `!=`	Compare ranges, ordering by start then width
	`sort`, `order`, `rank`	Sort by the ordering
	`duplicated`	Find ranges with multiple instances
	`unique`	Find unique instances, removing duplicates
Arithmetic	`r + x`, `r - x`, `r * x`	Shrink or expand ranges `r` by number `x`
	`shift`	Move the ranges by specified amount
	`resize`	Change width, anchoring on start, end or mid
	`distance`	Separation between ranges (closest endpoints)
	`restrict`	Clamp ranges to within some start and end
	`flank`	Generate adjacent regions on start or end
Set operations	`reduce`	Merge overlapping and adjacent ranges
	`intersect`, `union`, `setdiff`	Set operations on reduced ranges
	`pintersect`, `punion`, `psetdiff`	Parallel set operations, on each `x[i]`, `y[i]`
	`gaps`, `pgap`	Find regions not covered by reduced ranges
	`disjoin`	Ranges formed from union of endpoints
Overlaps	`findOverlaps`	Find all overlaps for each `x` in `y`
	`countOverlaps`	Count overlaps of each `x` range in `y`
	`nearest`	Find nearest neighbors (closest endpoints)
	`precede`, `follow`	Find nearest `y` that `x` precedes or follows
	`x %in% y`	Find ranges in `x` that overlap range in `y`
Coverage	`coverage`	Count ranges covering each position
Extraction	`r[i]`	Get or set by logical or numeric index
	`r[[i]]`	Get integer sequence from `start[i]` to `end[i]`
	`subsetByOverlaps`	Subset `x` for those that overlap in `y`
	`head`, `tail`, `rev`, `rep`	Conventional R semantics
Split, combine	`split`	Split ranges by a factor into a *RangesList*
	`c`	Concatenate two or more range objects

```
integer-Rle of length 28 with 12 runs
  Lengths: 6 2 4 1 2 3 3 1 1 3 1 1
  Values : 0 1 2 1 2 1 0 1 2 3 2 1
```

The run-length encoding can be interpreted as ‘a run of length 6 of nucleotides covered by 0 ranges, followed by a run of length 2 of nucleotides covered by 1 range…’.

Between methods act on two (or sometimes more) *IRanges* instances. These include `intersect`, `setdiff`, `union`, `pintersect`, `psetdiff`, and `punion`.

The `countOverlaps` and `findOverlaps` functions operate on two sets of ranges. `countOverlaps` takes its first argument (the `query`) and determines how many of the ranges in the second argument (the `subject`) each overlaps. The result is an integer vector with one element for each member of `query`. `findOverlaps` performs a similar operation but returns a more general matrix-like structure that identifies each pair of query / subject overlaps. Both arguments allow some flexibility in the definition of ‘overlap’.

Common operations on ranges are summarized in Table 2.8.

`mcols` **and** `metadata` The *GRanges* class (actually, most of the data structures defined or extending those in the *IRanges* package) has two additional very useful data components. The `mcols` function allows information on each range to be stored and manipulated (e.g., subset) along with the *GRanges* instance. The element metadata is represented as a *DataFrame*, defined in *IRanges* and acting like a standard *R data.frame* but with the ability to hold more complicated data structures as columns (and with element metadata of its own, providing an enhanced alternative to the *Biobase* class *AnnotatedDataFrame*).

```
> mcols(genes) <- DataFrame(EntrezId=c("42865", "2768869"),
+                           Symbol=c("kal-1", "CG34330"))
```

`metadata` allows addition of information to the entire object. The information is in the form of a list; any data can be provided.

```
> metadata(genes) <-
+     list(CreatedBy="A. User", Date=date())
```

GRangesList The `GRanges` class is extremely useful for representing simple ranges. Some next-generation sequence data and genomic features are more hierarchically structured. A gene may be represented by several exons within it. An aligned read may be represented by discontinuous ranges of alignment to a reference. The *GRangesList* class represents this type of information. It is a list-like data structure, which each element of the list itself a *GRanges* instance. The gene FBgn0039155 contains several exons, and can be represented as a list of length 1, where the element of the list contains a *GRanges* object with 7 elements:

```
GRangesList of length 1:
$FBgn0039155
GRanges with 7 ranges and 2 metadata columns:
      seqnames               ranges strand |   exon_id   exon_name
         <Rle>            <IRanges>  <Rle> | <integer> <character>
  [1]    chr3R [19967117, 19967382]      + |     50486        <NA>
  [2]    chr3R [19970915, 19971592]      + |     50487        <NA>
  [3]    chr3R [19971652, 19971770]      + |     50488        <NA>
  [4]    chr3R [19971831, 19972024]      + |     50489        <NA>
  [5]    chr3R [19972088, 19972461]      + |     50490        <NA>
  [6]    chr3R [19972523, 19972589]      + |     50491        <NA>
  [7]    chr3R [19972918, 19973212]      + |     50492        <NA>

---
seqlengths:
    chr3R
 27905053
```

The *GRangesList* object has methods one would expect for lists (e.g., `length`, sub-setting). Many of the methods introduced for working with *IRanges* are also available, with the method applied element-wise.

The GenomicFeatures package Many public resources provide annotations about genomic features. For instance, the UCSC genome browser maintains the 'knownGene' track of established exons, transcripts, and coding sequences of many model organisms. The *GenomicFeatures* package provides a way to retrieve, save, and query these resources. The underlying representation is as sqlite data bases, but the data are available in *R* as *GRangesList* objects. The following exercise explores the *GenomicFeatures* package and some of the functionality for the *IRanges* family introduced above.

Exercise 10
Load the TxDb.Dmelanogaster.UCSC.dm3.ensGene annotation package, and create an alias `txdb` *pointing to the TranscriptDb object this class defines.*

Extract all exon coordinates, organized by gene, using `exonsBy`*. What is the class of this object? How many elements are in the object? What does each element correspond to? And the elements of each element? Use* `elementLengths` *and* `table` *to summarize the number of exons in each gene, for instance, how many single-exon genes are there?*

Select just those elements corresponding to flybase gene ids FBgn0002183, FBgn0003360, FBgn0025111, and FBgn0036449. Use `reduce` *to simplify gene models, so that exons that overlap are considered 'the same'.*

Solution:

```
> library(TxDb.Dmelanogaster.UCSC.dm3.ensGene)
> txdb <- TxDb.Dmelanogaster.UCSC.dm3.ensGene # alias
> ex0 <- exonsBy(txdb, "gene")
> head(table(elementLengths(ex0)))
```

```
   1    2    3    4    5    6 
3161 2845 2157 1678 1172  929 
```

```
> ids <- c("FBgn0002183", "FBgn0003360", "FBgn0025111", "FBgn0036449")
> ex <- reduce(ex0[ids])
```

2.4.2 Alignments and the Rsamtools Package

Most down-stream analysis of short read sequences is based on reads aligned to reference genomes.

Alignment formats Most main-stream aligners produce output in SAM (text-based) or BAM format. A SAM file is a text file, with one line per aligned read, and fields separated by tabs. Here is an example of a single SAM line, split into fields.

```
> fl <- system.file("extdata", "ex1.sam", package="Rsamtools")
> strsplit(readLines(fl, 1), "\t")[[1]]
```

```
 [1] "B7_591:4:96:693:509"
 [2] "73"
 [3] "seq1"
 [4] "1"
 [5] "99"
 [6] "36M"
 [7] "*"
 [8] "0"
 [9] "0"
[10] "CACTAGTGGCTCATTGTAAATGTGTGGTTTAACTCG"
[11] "<<<<<<<<<<<<<<<;<<<<<<<<<5<<<<<;:<;7"
[12] "MF:i:18"
[13] "Aq:i:73"
[14] "NM:i:0"
[15] "UQ:i:0"
[16] "H0:i:1"
[17] "H1:i:0"
```

Fields in a SAM file are summarized in Table 2.9. We recognize from the FASTQ file the identifier string, read sequence and quality. The alignment is to a chromosome 'seq1' starting at position 1. The strand of alignment is encoded in the 'flag' field. The alignment record also includes a measure of mapping quality, and a CIGAR string describing the nature of the alignment. In this case, the CIGAR is 36M, indicating that the alignment consisted of 36 `M`atches or mismatches, with no indels or gaps; indels are represented by `I` and `D`; gaps (e.g., from alignments spanning introns) by `N`.

BAM files encode the same information as SAM files, but in a format that is more efficiently parsed by software; BAM files are the primary way in which aligned reads are imported in to *R*.

Aligned reads in R Important classes and methods for working with aligned reads are summarized in Table 2.10. The `readGAlignments` function from the *GenomicRanges* package reads essential information from a BAM file in to *R*. The result is an instance of the *GAlignments* class. The *GAlignments* class has been designed to allow useful manipulation of many reads (e.g., 20 million) under moderate memory requirements (e.g., 4 GB).

Table 2.9: Fields in a SAM record. From `http://samtools.sourceforge.net/samtools.shtml`

Field	Name	Value
1	QNAME	Query (read) NAME
2	FLAG	Bitwise FLAG, e.g., strand of alignment
3	RNAME	Reference sequence NAME
4	POS	1-based leftmost POSition of sequence
5	MAPQ	MAPping Quality (Phred-scaled)
6	CIGAR	Extended CIGAR string
7	MRNM	Mate Reference sequence NaMe
8	MPOS	1-based Mate POSition
9	ISIZE	Inferred insert SIZE
10	SEQ	Query SEQuence on the reference strand
11	QUAL	Query QUALity
12+	OPT	OPTional fields, format TAG:VTYPE:VALUE

Table 2.10: Input, representation, and manipulation of aligned reads from the *Rsamtools* and *GenomicRanges* packages; most *GRanges* operations are also supported.

Category	Function	Description
Classes	*GAlignments*	Single-end reads, see `?GAlignments`
	GAlignmentPairs	Strict paired-end reads, `?GAlignmentPairs`
	GAlignmentsList	Paired-end reads, `?GAlignmentsList`
	BamFile, *BamFileList*	BAM file reference
	ScanBamParam	Select regions and fields for input, `?ScanBamParam`
Input	`scanBamHeader`	Header summary
	`seqinfo`	'rname's and lengths
	`readGAlignments*`	Simple BAM file input
	`readBamGAlignments*`	Selective BAM input and iteration
	`scanBam`	'Low-level' input
	`applyPileups`	Per-nucleotide iteration
File manipulation		
	`countBam`, `quickCountBam`	Summarize file content
	`filterBam`	Filter one BAM file to another
	`sortBam`	Sort by position or 'qname'
	`indexBam`	Create an index

```
> alnFile <- system.file("extdata", "ex1.bam", package="Rsamtools")
> aln <- readGAlignments(alnFile)
> head(aln, 3)

GAlignments with 3 alignments and 0 metadata columns:
      seqnames strand       cigar    qwidth     start       end     width
         <Rle>  <Rle> <character> <integer> <integer> <integer> <integer>
  [1]     seq1      +         36M        36         1        36        36
  [2]     seq1      +         35M        35         3        37        35
  [3]     seq1      +         35M        35         5        39        35
           ngap
      <integer>
  [1]         0
  [2]         0
  [3]         0
  ---
  seqlengths:
   seq1 seq2
   1575 1584
```

The `readGAlignments` function takes an additional argument, `param`, allowing the user to specify regions of the BAM file (e.g., known gene coordinates) from which to extract alignments.

A *GAlignments* instance is like a data frame, but with accessors as suggested by the column names. It is easy to query, e.g., the distribution of reads aligning to each strand, the width of reads, or the cigar strings

```
> table(strand(aln))

   +    -    *
1647 1624    0

> table(width(aln))

  30   31   32   33   34   35   36   38   40
   2   21    1    8   37 2804  285    1  112

> head(sort(table(cigar(aln)), decreasing=TRUE))

     35M      36M      40M      34M      33M 14M4I17M
    2804      283      112       37        6        4
```

Exercise 11

Use `bigdata`*,* `file.path` *and* `dir` *to obtain file paths to the BAM files. These are a subset of the aligned reads, overlapping just four genes.*

Input the aligned reads from one file using `readGAlignments`*. Explore the reads, e.g., using* `table` *or* `xtabs`*, to summarize which chromosome and strand the subset of reads is from.*

The object ex created earlier contains coordinates of four genes. Use `countOverlaps` *to first determine the number of genes an individual read aligns to, and then the number of uniquely aligning reads overlapping each gene. Since the RNAseq protocol was not strand-sensitive, set the strand of* `aln` *to* `*`*.*

Write the sequence of steps required to calculate counts as a simple function, and calculate counts on each file. On Mac or Linux, can you easily parallelize this operation?

Solution: We discover the location of files using standard *R* commands:

```
> bigdata <- system.file("bigdata", package="SequenceAnalysisData")
> fls <- dir(file.path(bigdata, "bam"), ".bam$", full=TRUE) #$
> names(fls) <- sub("_.*", "", basename(fls))
```

Use `readGAlignments` to input data from one of the files, and standard *R* commands to explore the data.

```
> ## input
> aln <- readGAlignments(fls[1])
> xtabs(~seqnames + strand, as.data.frame(aln))

        strand
seqnames    +    -
   chr3L 5402 5974
   chrX  2278 2283
```

To count overlaps in regions defined in a previous exercise, load the regions.

```
> data(ex)              # from an earlier exercise
```

Many RNA-seq protocols are not strand aware, i.e., reads align to the plus or minus strand regardless of the strand on which the corresponding gene is encoded. Adjust the strand of the aligned reads to indicate that the strand is not known.

```
> strand(aln) <- "*"   # protocol not strand-aware
```

One important issue when counting reads is to make sure that the reference names in both the annotation and the read files are identical.

Exercise 12
Check the reference name in both the `ex` *and* `aln`*. If they are not similar, how could you correct them?*

For simplicity, we are interested in reads that align to only a single gene. Count the number of genes a read aligns to. . .

```
> hits <- countOverlaps(aln, ex)
> table(hits)
```

```
hits
    0     1     2
  772 15026   139
```

and reverse the operation to count the number of times each region of interest aligns to a uniquely overlapping alignment.

```
> cnt <- countOverlaps(ex, aln[hits==1])
```

A simple function for counting reads (see `GenomicRangese::summarizeOverlaps` for greater detail) is

```
> counter <-
+     function(filePath, range)
+ {
+     aln <- readGAlignments(filePath)
+     strand(aln) <- "*"
+     hits <- countOverlaps(aln, range)
+     countOverlaps(range, aln[hits==1])
+ }
```

This can be applied to all files using `sapply`

```
> counts <- sapply(fls, counter, ex)
```

The counts in one BAM file are independent of counts in another BAM file. This encourages us to count reads in each BAM file in parallel, decreasing the length of time required to execute our program. On Linux and Mac OS, a straight-forward way to parallelize this operation is:

```
> if (require(parallel))
+     simplify2array(mclapply(fls, counter, ex))
```

2.5 Integrating data and annotations: SummarizedExperiment

The *SummarizedExperiment* class represents one way to integrate information on samples (e.g., covariates, treatment group), ranges of interest, and derived measurements. This class contains a list of `assay` matrix elements, each with the same dimension and corresponding to an analysis performed on regions of interest (`rowData`, represented by a *GRanges* or *GRangesList* instance) across samples (`colData`, a *DataFrame* describing sample attributes). The *SummarizedExperiment* class is designed so that it behaves in important ways like a *GenomicRanges* instance, e.g., `subsetByOverlaps` can be used to select the subset of rows in the *SummarizedExperiment* instance that overlap annotated regions of interest. By tightly coupling the row and column annotations with the assay data, one reduces the consequences of clerical or other errors and greatly facilitates development of reproducible work flows.

Chapter 3

Working with large data

Bioinformatics data is now very large; it is not reasonable to expect all of a FASTQ or BAM file, for instance, to fit into memory. How is this data to be processed? This challenge confronts us in whatever language or tool we are using. In *R* and *Bioconductor*, the main approaches are to: (1) write *efficient* *R* code; (2) *restrict* data input to the interesting subset of the larger data set; (3) *sample* from the large data, knowing that an appropriately sized sample will accurately estimate statistics we are interested in; (4) *iterate* through large data in chunks; and (5) use *parallel* evaluation on one or several computers. Let's look at each of these approaches.

3.1 Efficient R code

There are often many ways to accomplish a result in *R*, but these different ways often have very different speed or memory requirements. For small data sets these performance differences are not that important, but for large data sets (e.g., high-throughput sequencing; genome-wide association studies, GWAS) or complicated calculations (e.g., bootstrapping) performance can be important. Several approaches to achieving efficient *R* programming are summarized in Table 3.1; common tools used to help with assessing performance (including comparison of results from different implementations!) are in Table 3.2.

Easy solutions Several common performance bottlenecks often have easy solutions; these are outlined here.

Table 3.1: Common ways to improve efficiency of *R* code.

Easy	Moderate
1. Selective input	1. Know relevant packages
2. Vectorize	2. Understand algorithm complexity
3. Pre-allocate and fill	3. Use parallel evaluation
4. Avoid expensive conveniences	4. Exploit libraries and `C++` code

Table 3.2: Tools for measuring performance.

Function	Description
`identical` `all.equal`	Compare content of objects.
`system.time`	Time required to evaluate an expression
`Rprof`	Time spent in each function; also `summaryRprof`.
`tracemem`	Indicate when memory copies occur (*R* must be configured to support this).
rbenchmark *microbenchmark*	Packages for standardizing speed measurment

Text files often contain more information, for example 1000's of individuals at millions of SNPs, when only a subset of the data is required, e.g., during algorithm development. Reading in all the data can be demanding in terms of both memory and time. A solution is to use arguments such as `colClasses` to specify the columns and their data types that are required, and to use `nrow` to limit the number of rows input. For example, the following ignores the first and fourth column, reading in only the second and third (as type `integer` and `numeric`).

```
> ## not evaluated
> colClasses <-
+   c("NULL", "integer", "numeric", "NULL")
> df <- read.table("myfile", colClasses=colClasses)
```

R is vectorized, so traditional programming `for` loops are often not necessary. Rather than calculating 100000 random numbers one at a time, or squaring each element of a vector, or iterating over rows and columns in a matrix to calculate row sums, invoke the single function that performs each of these operations.

```
> x <- runif(100000); x2 <- x^2
> m <- matrix(x2, nrow=1000); y <- rowSums(m)
```

This often requires a change of thinking, turning the sequence of operations 'inside-out'. For instance, calculate the log of the square of each element of a vector by calculating the square of all elements, followed by the log of all elements `x2 <- x^2; x3 <- log(x2)`, or simply `logx2 <- log(x^2)`.

It may sometimes be natural to formulate a problem as a `for` loop, or the formulation of the problem may require that a `for` loop be used. In these circumstances the appropriate strategy is to pre-allocate the `result` object, and to fill the result in during loop iteration.

```
> ## not evaluated
> result <- numeric(nrow(df))
> for (i in seq_len(nrow(df)))
+   result[[i]] <- some_calc(df[i,])
```

Failure to pre-allocate and fill is the second cirle of *R* hell [6].

Some *R* operations are helpful in general, but misleading or inefficient in particular circumstances. An example is the behavior of `unlist` when the list is named – *R* creates new names that have been made unique. This can be confusing (e.g., when Entrez gene identifiers are 'mangled' to unintentionally look like other identifiers) and expensive (when a large number of new names need to be created). Avoid creating unnecessary names, e.g.,

```
> unlist(list(a=1:2)) # name 'a' becomes 'a1', 'a2'

a1 a2
 1  2

> unlist(list(a=1:2), use.names=FALSE)   # no names

[1] 1 2
```

Names can be very useful for avoiding book-keeping errors, but are inefficient for repeated look-ups; use vectorized access or numeric indexing.

Moderate solutions Several solutions to inefficient code require greater knowledge to implement.

Using appropriate functions can greatly influence performance; it takes experience to know when an appropriate function exists. For instance, the `lm` function could be used to assess differential expression of each gene on a microarray, but the *limma* package implements this operation in a way that takes advantage of the experimental design that is common to each probe on the microarray, and does so in a very efficient manner.

```
> ## not evaluated
> library(limma) # microarray linear models
> fit <- lmFit(eSet, design)
```

Using appropriate algorithms can have significant performance benefits, especially as data becomes larger. This solution requires moderate skills, because one has to be able to think about the complexity (e.g., expected number of operations) of an algorithm, and to identify algorithms that accomplish the same goal in fewer steps. For example, a naive way of identifying which of 100 numbers are in a set of size 10 might look at all 100×10 combinations of numbers (i.e., polynomial time), but a faster way is to create a 'hash' table of one of the set of elements and probe that for each of the other elements (i.e., linear time). The latter strategy is illustrated with

```
> x <- 1:100; s <- sample(x, 10)
> inS <- x %in% s
```

R supports parallel evaluation, most easily through the `mclapply` function of the *parallel* package distributed with base *R* (`mclapply` is unfortunately not available on Windows). Parallel evaluation is discussed further in Chapter 3.

R is an interpreted language, and for very challenging computational problems it may be appropriate to write critical stages of an analysis in a compiled language like C or Fortran, or to use an existing programming library (e.g., the BOOST library) that efficiently implements advanced algorithms. *R* has a well-developed interface to C or Fortran, so it is 'easy' to do this; the *Rcpp* package provides a very nice approach for those familiar with C++ concepts. This places a significant burden on the person implementing the solution, requiring knowledge of two or more computer languages and of the interface between them.

Measuring performance When trying to improve performance, one wants to ensure (a) that the new code is actually faster than the previous code, and (b) both solutions arrive at the same, correct, answer.

The `system.time` function is a straight-forward way to measure the length of time a portion of code takes to evaluate.

```
> m <- matrix(runif(200000), 20000)
> system.time(apply(m, 1, sum))
```

```
   user  system elapsed 
  0.048   0.000   0.046 
```

When comparing performance of different functions, it is appropriate to replicate timings to average over vagaries of system use, and to shuffle the order in which timings of alternative algorithms are calculated to avoid artifacts such as initial memory allocation. Rather than creating *ad hoc* approaches to timing, it is convenient to use packages such as *rbenchmark*:

```
> library(rbenchmark)
> f0 <- function(x) apply(x, 1, sum)
> f1 <- function(x) rowSums(x)
> benchmark(f0(m), f1(m),
+           columns=c("test", "elapsed", "relative"),
+           replications=5)
```

```
   test elapsed relative
1 f0(m)   0.241   80.333
2 f1(m)   0.003    1.000
```

Speed is an important metric, but equivalent results are also needed. The functions `identical` and `all.equal` provide different levels of assessing equivalence, with `all.equal` providing ability to ignore some differences, e.g., in the names of vector elements.

```
> res1 <- apply(m, 1, sum)
> res2 <- rowSums(m)
> identical(res1, res2)
```

```
[1] TRUE
```

```
> identical(c(1, -1), c(x=1, y=-1))
```

```
[1] FALSE
```

```
> all.equal(c(1, -1), c(x=1, y=-1),
+           check.attributes=FALSE)

[1] TRUE
```

Two additional functions for assessing performance are `Rprof` and `tracemem`; these are mentioned only briefly here. The `Rprof` function profiles *R* code, presenting a summary of the time spent in each part of several lines of *R* code. It is useful for gaining insight into the location of performance bottlenecks when these are not readily apparent from direct inspection. Memory management, especially copying large objects, can frequently contribute to poor performance. The `tracemem` function allows one to gain insight into how *R* manages memory; insights from this kind of analysis can sometimes be useful in restructuring code into a more efficient sequence.

3.2 Restriction

Just because a data file contains a lot of data does not mean that we are interested in all of it. In base *R*, one might use the `colClasses` argument to `read.delim` or similar function (e.g., setting some elements to `NULL`) to read only some columns of a large comma-separated value file. In addition to the obvious benefit of using less memory than if all of the file had been read in, input will be substantially faster because less computation needs to be done to coerce values from their representation in the file to their representation in *R*'s memory.

A variation on the idea of restricting data input is to organize the data on disk into a representation that facilitates restriction. In base *R*, large data might be stored in a relational data base like the *sqlite* data bases that are built in to *R* and used in the *AnnotationDbi Bioconductor* packages. In addition to facilitating restriction, these approaches are typically faster than parsing a plain text file, because the data base software has stored data in a way that efficiently transforms from on-disk to in-memory representation.

Restriction is such a useful concept that many *Bioconductor* high throughput sequence analysis functions enable doing the right thing. Functions such as `coverage,BamFile-method` use restrictions to read in the specific data required for them to compute the statistic of interest. Most "higher level" functions have a `param=ScanBamParam()` argument. For instance, the primary user-friendly function for reading BAM files is `readGAlignments`. This function reads the most useful information, allowing the user to specify additional fields if desired.

```
> fls <- RNAseqData.HNRNPC.bam.chr14_BAMFILES # 8 BAM files
> bamfls <- BamFileList(fls, yieldSize=500000) # yieldSize can be larger
> gal <- readGAlignments(bamfls[[1]])
> head(gal, 3)

GAlignments with 3 alignments and 0 metadata columns:
      seqnames strand       cigar    qwidth     start       end     width
         <Rle>  <Rle> <character> <integer> <integer> <integer> <integer>
  [1]    chr14      +         72M        72  19069583  19069654        72
  [2]    chr14      +         72M        72  19363738  19363809        72
  [3]    chr14      -         72M        72  19363755  19363826        72
           ngap
      <integer>
  [1]         0
  [2]         0
  [3]         0
  ---
  seqlengths:
                  chr1                chr10 ...                 chrY
             249250621            135534747 ...             59373566

> param <- ScanBamParam(what="seq")  ## also input sequence
> gal <- readGAlignments(bamfls[[1]], param=param)
> head(mcols(gal)$seq)
```

```
  A DNAStringSet instance of length 6
    width seq
[1]    72 TGAGAATGATGATTTCCAATTTCATCCATGTCCC...AGGACATGAACTCATCATTTTTTATGGCTGCAT
[2]    72 CCCATATGTACATCAGGCCCCAGGTATACACTGG...AGGTGGACACCAGCACTCAGTTGGATACACACA
[3]    72 CCCCAGGTATACACTGGACTCCAGGTGGACACCA...CAGTTGGATACACACACTCAAGGTGGACACCAG
[4]    72 CATAGATGCAAGAATCCTCAATCAAATACTAGCA...AATTCAACAGCACATTAAAAAGATAACTTACCA
[5]    72 TAGCACACTGAATTCAACAGCACATTAAAAAGAT...ACCATGCTCAAGTGGATTTACCCCAAGGATACA
[6]    72 TGCTGGTGCAGGATTTATTCTACTAAGCAATGAG...GGATCAAATCCACTTTCTTATCTCAGGAATCAG
```

Exercise 13

Let's use BAM files and the Rsamtools package to illustrate restriction. Rather than using simple text files ("sam" format), we use "bam" (binary alignment) files that have been indexed. Use of a binary format enhances data input speed, while the index facilitates restrictions that take into account genomic coordinates. The basic approach will use the `ScanBamParam` *function to specify a restriction.*

A BAM record can contain a lot of information, including the relatively large sequence, quality, and query name strings. Not all information in the BAM file is needed for some calculations. For instance, one could calculate coverage (number of nucleotides overlapping each reference position) using only the `rname` *(reference sequence name),* `pos`*, and* `cigar` *fields. We could arrange to input this just information with*

```
> param <- ScanBamParam(what=c("rname", "pos", "cigar"))
```

Another common restriction is to particular genomic regions, for instance known genes in an RNAseq differential expression study or to promoter regions in a ChIP-seq study. Restrictions in genome space are specified using `GRanges` *objects (typically computed from some reference source, rather than entered by hand) provided as the* `which` *argument to* `ScanBamParam`*. Here we create a* `GRanges` *instance representing all exons on* D. melanogaster *chromosome 3L, and use that as a restriction to* `ScanBamParam`*'s* `which` *argument:*

```
> library(TxDb.Dmelanogaster.UCSC.dm3.ensGene)
> exByGn <- exonsBy(TxDb.Dmelanogaster.UCSC.dm3.ensGene, "gene")
> seqlevels(exByGn, force=TRUE) <- "chr3L"
> gns <- unlist(range(exByGn))
> param <- ScanBamParam(which=gns)
```

The `what` *and* `which` *restrictions can be combined with other restrictions into a single* `ScanBamParam` *object*

```
> param <- ScanBamParam(what=c("rname", "pos", "cigar"), which=gns)
```

We can also restrict input to, e.g., paired-end reads that represent the primary (best) alignment; see the help page `?ScanBamParam` *for a more complete description.*

Reaching ahead a little bit, here are some BAM files from an experiment in D. melanogaster*; the BAM files contain a subset of aligned reads*

```
> bigdata <- system.file("bigdata", package="SequenceAnalysisData")
> bamfls <- dir(file.path(bigdata, "bam"), ".bam$", full=TRUE) #$
> names(bamfls) <- sub("_.*", "", basename(bamfls))
```

We'll read in the first BAM file, but restrict input to "rname" to find how many reads map to each chromosome. We use the "low-level" function `scanBam` *to input the data*

```
> param <- ScanBamParam(what="rname")
> bam <- scanBam(bamfls[1], param=param)[[1]]
> table(bam$rname)

  chr2L   chr2R   chr3L   chr3R    chr4    chrM    chrX chrYHet 
      0       0   11376       0       0       0    4561       0 
```

3.3 Sampling

R is after all a statistical language, and it sometimes makes sense to draw inferences from a sample of large data. For instance, many quality assessment statistics summarize overall properties (e.g., GC content or per-nucleotide base quality of FASTQ reads) that don't require processing of the entire data. For these statistics to be valid, the sample from the file needs to be a random sample, rather than a sample of convenience.

There are two advantages to sampling from a FASTQ (or BAM) file. The sample uses less memory than the full data. And because less data needs to be parsed from the on-disk to in-memory representation the input is faster.

The *ShortRead* package `FastqSampler` (see also `Rsamtools::BamSampler` visits a FASTQ file but retains only a random sample from the file. Here is our function to calculate GC content from a `DNAStringSet`:

```
> gcFunction <-
+     function(x)
+ {
+     alf <- alphabetFrequency(x, as.prob=TRUE)
+     rowSums(alf[,c("G", "C")])
+ }
```

and a subset of a FASTQ file with 1 million reads:

```
> bigdata <- system.file("bigdata", package="SequenceAnalysisData")
> fqfl <- dir(file.path(bigdata, "fastq"), ".fastq$", full=TRUE) #$
```

`FastqSampler` works by specifying the file name and desired sample size, e.g., 100,000 reads. This creates an object from which an independent sample can be drawn using the `yield` function.

```
> sampler <- FastqSampler(fqfl, 100000)
> fq <- yield(sampler)                    # 100,000 reads
> lattice::densityplot(gcFunction(sread(fq)), plot.points=FALSE)
> fq <- yield(sampler)                    # a different 100,000 reads
```

Generally, one would choose a sample size large enough to adequately characterize the data but not so large as to consume all (or a fraction, see section 3.5 below) of the memory. The default (1 million) is a reasonable starting point.

`FastqSampler` relies on the *R* random number generator, so the same sequence of reads can be sampled by using the same random number seed. This is a convenient way to sample the same read pairs from the FASTQ files typically used to represent paired-end reads

```
> ## NOT RUN
> set.seed(123)
> end1 <- yield(FastqSampler("end_1.fastq"))
> set.seed(123)
> end2 <- yield(FastqSampler("end_2.fastq"))
```

3.4 Iteration

Restriction may not be enough to wrestle large data down to size, and sampling may be inappropriate for the task at hand. A solution is then to iterate through the file. An example in base *R* is to open a file connection, and then read and process successive chunks of the file, e.g., reading chunks of 10000 lines

```
> ## NOT RUN
> con <- file("<hypothetical-file>.txt")
> open(f)
> while (length(x <- readLines(f, n=10000))) {
+     ## work on character vector 'x'
+ }
> close(f)
```

The pattern length(x <- readLines(f, n=10000)) is a convenient short-hand pattern that reads the *next* 10000 lines into a variable x and then asks x it's length. x is created in the calling environment (so it is available for processing in the while loop) When there are no lines left to read, length(x) will evaluate to zero and the while loop will end.

Iteration really involves three steps: input of a 'chunk' of the data; calculation of a desired summary; and aggregation of summaries across chunks. We illustrate this with BAM files in the *Rsamtools* package; see also FastqStreamer in the *ShortRead* package and TabixFile for iterating through large VCF files (via readVCF in the *VariantAnnotation* package).

The first stage in iteration is to arrange for input of a chunk of data. In many programming languages one would iterate 'record-at-a-time', reading in one record, processing it, and moving to the next. *R* is not efficient when used in this fashion. Instead, we want to read in a larger chunk of data, typically as much as can comfortably fit in the available memory. In *Rsamtools* for reading BAM files we arrange for this by creating a BamFile (or TabixFile for VCF) object where we specify an appropriate yieldSize. Here we go for a bigger BAM file

```
> library(RNAseqData.HNRNPC.bam.chr14)
> bamfl <- RNAseqData.HNRNPC.bam.chr14_BAMFILES[1]
> countBam(bamfl)

  space start end width                file records nucleotides
1    NA    NA  NA    NA ERR127306_chr14.bam  800484    57634848

> bf <- BamFile(bamfl, yieldSize=200000) # could be larger, e.g., 2 million
```

A BamFile object can be used to read data from a BAM file, e.g., using readGAlignments. Instead of reading all the data, the reading function will read just yieldSize records. The idea then is to open the BAM file and iterate through until no records are input. The pattern is

```
> ## initialize, e.g., for step 3 ...
> open(bf)
> while (length(gal <- readGAlignments(bf))) {
+     ## step 2: do work...
+     ## step 3: aggregate results...
+ }
> close(bf)
```

The second step is to perform a useful calculation on the chunk of data. This is particularly easy to do if chunks are independent of one another. For instance, a common operation is to count the number of times reads overlap regions of interest. There are functions that implement a variety of counting modes (see, e.g., summarizeOverlaps in the *GenomicRanges* package); here we'll go for a simple counter that arranges to tally one 'hit' each time a read overlaps (in any way) at most one range. Here is our counter function, taking as arguments a GRanges or GRangesList object representing the regions for which counts are desired, and a GAlignments object representing reads to be counted:

```
> counter <-
+     function(query, subject, ..., ignore.strand=TRUE)
+     ## query: GRanges or GRangesList
+     ## subject: GAlignments
+ {
+     if (ignore.strand)
+         strand(subject) <- "*"
+     hits <- countOverlaps(subject, query)
+     countOverlaps(query, subject[hits==1])
+ }
```

To set this step up a little more completely, we need to know the regions over which counting is to occur. Here we retrieve exons grouped by gene for the genome to which the BAM files were aligned:

```
> library(TxDb.Hsapiens.UCSC.hg19.knownGene)
> query <- exonsBy(TxDb.Hsapiens.UCSC.hg19.knownGene, "gene")
```

Thus we'll add the following lines to our loop:

```
> ## initialize, e.g., for step 3 ...
> open(bf)
> while (length(gal <- readGAlignments(bf))) {
+     ## step 2: do work...
+     count0 <- counter(query, gal, ignore.strand=TRUE)
+     ## step 3: aggregate results...
+ }
> close(bf)
```

It is worth asking whether the 'do work' step will always be as straight-forward. The current example is easy because counting overlaps of one read does not depend on other reads, so the chunks can be processed independently of one another. *FIXME: These are paired-end reads, so counting is* not *this easy! See* `?summarizeOverlaps` *in the GenomicRanges package.*

The final step in iteration is to aggregate results across chunks. In our present case, `counter` always returns an integer vector with `length(query)` elements. We want to add these over chunks, and we can arrange to do this by starting with an initial vector of counts `counts <- integer(length(query))` and simply adding `count0` at each iteration. The complete iteration, packaged as a function to facilitate re-use is

```
> counter1 <-
+     function(bf, query, ...)
+ {
+     ## initialize, e.g., for step 3 ...
+     counts <- integer(length(query))
+     open(bf)
+     while (length(gal <- readGAlignments(bf, ...))) {
+         ## step 2: do work...
+         count0 <- counter(query, gal, ignore.strand=TRUE)
+         ## step 3: aggregate results...
+         counts <- counts + count0
+     }
+     close(bf)
+     counts
+ }
```

In action, this is invoked as

```
> bf <- BamFile(bamfl, yieldSize=500000)
> counts <- counter1(bf, query)
```

Counting is a particularly simple operation; one will often need to think carefully about how to aggregate statistics derived from individual chunks. A useful general approach is to identify *sufficient statistics* that represent a sufficient description of the data and can be easily aggregated across chunks. This is the approach taken by, for instance, the *biglm* package for fitting linear models to large data sets – the linear model is fit to successive chunks and the fit reduced to sufficient statistics, the sufficient statistics are then added across chunks to arrive at an overall fit.

3.5 Parallel evaluation

Each of the preceding sections addressed memory management; what about overall performance?

The starting point is really writing efficient, vectorized code, with common strategies outlined in Chapter 1. Performance differences between poorly written versus well written *R* code can easily span two orders of magnitude, whereas parallel processing can only increase throughput by an amount inversely proportional to the number of processing units (e.g., CPUs) available.

The memory management techniques outlined earlier in this chapter are important in a parallel evaluation context. This is because we will typically be trying to exploit multiple processing cores on a single computer, and the cores will be competing for the same pool of shared memory. We thus want to arrange for the collection of processors to cooperate in dividing available memory between them, i.e., each processor needs to use only a fraction of total memory.

There are a number of ways in which *R* code can be made to run in parallel. The least painful and most effective will use 'multicore' functionality provided by the *parallel* package; the *parallel* package is installed by default with base *R*, and has a useful vignette [30]. Unfortunately, multicore facilities are not available on Windows computers.

Parallel evaluation on several cores of a single Linux or MacOS computer is particularly easy to achieve when the code is already vectorized. The solution on these operating systems is to use the functions `mclapply` or `pvec`. These functions allow the 'master' process to 'fork' processes for parallel evaluation on each of the cores of a single machine. The forked processes initially share memory with the master process, and only make copies when the forked process modifies a memory location ('copy on change' semantics). On Linux and MacOS, the `mclapply` function is meant to be a 'drop-in' replacement for `lapply`, but with iterations being evaluated on different cores. The following illustrates the use of `mclapply` and `pvec`, for a toy vectorized function `f`:

```
> library(parallel)
> f <- function(i) {
+     cat("'f' called, length(i) = ", length(i), "\n")
+     sqrt(i)
+ }
> res0 <- mclapply(1:5, f, mc.cores=2)
```

```
'f' called, length(i) =  1
'f' called, length(i) =  1
'f' called, length(i) =  1
'f' called, length(i) =  1
'f' called, length(i) =  1
```

```
> res1 <- pvec(1:5, f, mc.cores=2)
```

```
'f' called, length(i) =  3
'f' called, length(i) =  2
```

```
> identical(unlist(res0), res1)
```

```
[1] TRUE
```

`pvec` takes a vectorized function and distributes computation of different chunks of the vector across cores. Both functions allow the user to specify the number of cores used, and how the data are divided into chunks.

The *parallel* package does not support fork-like behavior on Windows, where users need to more explicitly create a cluster of *R* workers and arrange for each to have the same data loaded into memory; similarly, parallel evaluation across computers (e.g., in a cluster) require more elaborate efforts to coordinate workers; this is typically done using `lapply`-like functions provided by the *parallel* package but specialized for simple ('snow') or more robust ('MPI') communication protocols between workers.

Data movement and *random numbers* are two important additional considerations in parallel evaluation. Moving data to and from cores to the manager can be expensive, so strategies that minimize explicit movement (e.g., passing file names data base queries rather than *R* objects read from files; reducing data on the worker before transmitting results to the manager) can be important. Random numbers need to be synchronized across cores to avoid generating the same sequences on each 'independent' computation.

How might parallel evaluation be exploited in *Bioconductor* work flows? One approach when working with BAM files exploits the fact that data are often organized with one sample per BAM file. Suppose we are interested in running our iterating counter `counter1` over several BAM files. We could do this by creating a `BamFileList` with appropriate `yieldSize`

```
> fls <- RNAseqData.HNRNPC.bam.chr14_BAMFILES # 8 BAM files
> bamfls <- BamFileList(fls, yieldSize=500000) # yieldSize can be larger
```

and using an `lapply` to take each file in turn and performing the count.

```
> counts <- simplify2array(lapply(bamfls, counter1, query))
```

The parallel equivalent of this is simply (note the change from `lapply` to `mclapply`)

```
> options(mc.cores=detectCores())         # use all cores
> counts <- simplify2array(mclapply(bamfls, counter1, query))
> head(counts[rowSums(counts) != 0,], 3)
```

```
          ERR127306 ERR127307 ERR127308 ERR127309 ERR127302 ERR127303 ERR127304
10001           207       277       217       249       303       335       362
100128927       572       629       597       483       379       319       388
100129075        62        70        56        63        34        51        88
          ERR127305
10001           300
100128927       435
100129075        79
```

3.6 And. . .

The *foreach* package can be useful for parallel evaluation written using coding styles more like `for` loops rather than `lapply`. The *iterator* package is an abstraction that simplifies the notion of iterating over objects. The *Rmpi* package provides access to MPI, a structured environment for calculation on clusters. The *pbdR* formalism[1] is especially useful for well-structured distributed matrix computations.

The *MatrixEQTL* package is an amazing example implementing high performance algorithms on large data; the corresponding publication [35] is well worth studying.

Bioconductor provides an Amazon machine instance with MPI and *Rmpi* installed[2]. This can be an effective way to gain access to large computing resources.

[1] https://rdav.nics.tennessee.edu/2012/09/pbdr/

[2] http://bioconductor.org/help/bioconductor-cloud-ami/

Part II

Differential Representation

Chapter 4

Statistical Issues in High-Throughput Sequence Analysis

4.1 General work flows

A running example: the pasilla data set As a running example, we use the *pasilla* data set, derived from [4]. The authors investigate conservation of RNA regulation between *D. melanogaster* and mammals. Part of their study used RNAi and RNA-seq to identify exons regulated by Pasilla (*ps*), the *D. melanogaster* ortholog of mammalian NOVA1 and NOVA2. Briefly, their experiment compared gene expression as measured by RNAseq in S2-DRSC cells cultured with, or without, a 444 bp dsRNA fragment corresponding to the *ps* mRNA sequence. Their assessment investigated differential exon use, but our worked example will focus on gene-level differences. For several examples we look at a subset of the *ps* data, corresponding to reads obtained from lanes of their RNA-seq experiment, and to the same reads aligned to a *D. melanogaster* reference genome. Reads were obtained from GEO and the Short Read Archive (SRA), and were aligned to the *D. melanogaster* reference genome *dm3* as described in the *pasilla* experiment data package.

Work flow At a very high level, one can envision a work flow that starts with a challenging biological question (how does *ps* influence gene and transcript regulation?). The biological question is framed in terms of wet-lab protocols coupled with an appropriate and all-important experimental design. There are several well-known statistical challenges, common to any experimental data What treatments are going to be applied? How many replicates will there be of each? Is there likely to be sufficient power to answer the biologically relevant question? Reality is also important at this stage, as evidenced in the *pasilla* data where, as we will see, samples were collected using different methods (single versus paired end reads) over a time when there were rapid technological changes. Such reality often introduces confounding factors that require appropriate statistical treatment in subsequent analysis.

The work flow proceeds with data generation, involving both a wet-lab (sample preparation) component and actual sequencing. It is essential to acknowledge the biases and artifacts that are introduced at each of these stages. Sample preparation involves non-trivial amounts of time and effort. Large studies are likely to have batch effects (e.g., because work was done by different lab members, or different batches of reagent). Samples might have been prepared in ways that are likely to influence down-stream analysis, e.g., using a protocol involving PCR and hence introducing opportunities for sample-specific bias. DNA isolation protocols may introduce many artifacts, e.g., non-uniform representation of reads across the length of expressed genes in RNA-seq. The sequencing reaction itself is far from bias-free, with known artifacts of called base frequency, cycle-dependent accuracy and bias, non-uniform coverage, etc. At a minimum, the research needs to be aware of the opportunities for bias that can be introduced during sample preparation and sequencing.

The informatics component of work flows becomes increasing important during and after sequence generation. The sequencer is often treated as a 'black box', producing short reads consisting of 10's to 100's of nucleotides and with associated quality scores. Usually, the chemistry and informatics processing pipeline are sufficiently well documented that one can arrive at an understanding of biases and quality issues that might be involved; such an understanding is likely to be particularly important when embarking on questions or using protocols that are at the fringe of standard practice (where, after all, the excitement is).

The first real data seen by users are *fastq* files (Table 2.3). These files are often simple text files consisting of many millions of records, and are described in greater detail in Section 2.3.2. The center performing the sequencing typically vets results for quality, but these quality measures are really about the performance of their machines. It is very important to assess quality with respect to the experiment being undertaken – Are the numbers of reads consistent across samples? Is the GC content and other observable aspects of the reads consistent with expectation? Are there anomalies in the sequence results that reflect primers or other reagents used during sample preparation? Are well-known artifacts of the protocol used evident in the reads in hand?

The next step in many work flows involves alignment of reads to a reference genome. There are many aligners available, including BWA [21], Bowtie / Bowtie2 [15], and GSNAP; merits of these are discussed in the literature. *Bioconductor* packages 'wrapping' these tools are increasingly common (e.g., *Rbowtie*, *gmapR*; *cummeRbund* for parsing output of the *cufflinks* transcript discovery pathway). There are also alignment algorithms implemented in *Bioconductor* (e.g., `matchPDict` in the *Biostrings* package, and the *Rsubread* package); `matchPDict` is particularly useful for flexible alignment of moderately sized subsets of data. Most main-stream aligners produce output in 'SAM' or 'BAM' (binary alignment) format. BAM files are the primary starting point for many analyses, and their manipulation and use in *Bioconductor* is introduced in Section 2.4.2.

4.2 RNA-seq: case study

4.2.1 Varieties of RNA-seq

RNA-seq experiments typically ask about differences in transcription of genes or other features across experimental groups. The analysis of designed experiments is statistical, and hence an ideal task for *R*. The overall structure of the analysis, with tens of thousands of features and tens of samples, is reminiscent of microarray analysis; some insights from the microarray domain will apply, at least conceptually, to the analysis of RNA-seq experiments.

The most straight-forward RNA-seq experiments quantify abundance for known gene models. The known models are derived from reference databases, reflecting the accumulated knowledge of the community responsible for the data. The 'knownGenes' track of the UCSC genome browser represents one source of such data. A track like this describes, for each gene, the transcripts and exons that are expected based on current data. The *GenomicFeatures* package allows ready access to this information by creating a local database out of the track information. This data base of known genes is coupled with high throughput sequence data by counting reads overlapping known genes and modeling the relationship between treatment groups and counts.

A more ambitious approach to RNA-seq attempts to identify novel transcripts. This requires that sequenced reads be assembled into contigs that, presumably, correspond to expressed transcripts that are then located in the genome. Regions identified in this way may correspond to known transcripts, to novel arrangements of known exons (e.g., through alternative splicing), or to completely novel constructs. We will not address the identification of completely novel transcripts here, but will instead focus on the analysis of the designed experiments: do the transcript abundances, novel or otherwise, differ between experimental groups?

4.2.2 RNA-seq work flows

RNA-seq work flows aim at measuring gene expression through assessment of mRNA abundance. Work flows involve:

1. Experimental design.
2. Wet-lab protocols for mRNA extraction and reverse transcription to cDNA.
3. Sequencing; QA.
4. Alignment of sequenced reads to a reference genome; QA.
5. Summarizing of the number of reads aligning to a region; QA.
6. Normalization of samples to accommodate purely technical differences in preparation.
7. Statistical assessment of differential representation, including specification of an appropriate error model.
8. Interpretation of results in the context of original biological questions; QA.

The inference is that higher levels of gene expression translate to more abundant cDNA, and greater numbers of reads aligned to the reference genome. The enumeration above seems simplistic, but oddly enough one has concerns and commentary on each point.

Table 4.1: Statistical issues in RNA-seq differential expression.

Analysis stage	Issues
Experimental design	Replication, complexity, feasibility
Batch effects	Known and unknown factors.
Summarize	Counts versus RPKM and other summaries.
Normalize	Robust estimates of library size.
Differential expression	Appropriate error model (Negative Binomial, Poisson, . . .); dispersion (under negative binomial) as parameter requiring estimation; 'shrinkage' to balance accuracy of per-gene estimates with precision of experiment-wide estimates.
Testing	Filtering to reduce multiple comparisons & false discovery rate.

4.2.3 Wet-lab protocols, sequencing, and alignment

The important point here is that wet-lab protocols, sequencing reactions, and alignment introduce artifacts that need to be acknowledged and, if possible, accommodated in down-stream analysis. These artifacts and approaches to their remediation are discussed in the following sections.

4.3 Statistical issues

Important statistical issues are summarized in Table 4.1.

4.3.1 Experimental design

Technical versus biological replication Obviously one should follow best practices for designing experiments appropriate for the data under analysis. A typical experiment will have one or several groups. Because there is uncertainty in each measurement, we require replication. Previous work shows that technical replication (repeating identical wet-lab and sequencing protocols on a single biological sample) introduces variation that is small [22] compared to biological replicates (using different samples). Most RNA-seq experiments require biological replication, and seldom include technical replicates.

Sample size How many biological replicates? It is helpful to think in terms of orders of magnitude – biological treatments with strong and consistent consequences for gene expression will be detected with a handful – 2 or 3 – replicates per treatment. Conversely, statistically subtle effects will not be much revealed by samples of say 5 or 8, but will instead require 10's or 100's of samples. The *RNASeqPower* package provides data-driven guidance on power calculations in RNA-seq experiments; *CSSP* provides ChIP-seq power calculations based on Bayesian estimation for local counting processes.

Complexity How complicated an experimental design? The advice must be to 'keep it simple'. There are many interesting biological questions that one could ask, but experimental designs with more than one or at most two factors, or with multiple levels per factor, will undermine statistical power and complicate analysis. There are exceptions of course, for instance a time course design or an experiment with two or more factors, but these require strong *a priori* motivation and confidence that the design is amenable to analysis even in the face of wet-lab or sequencing catastrophe.

Feasibility of intended statistical analysis What kind of treatment? Two 'lessons learned' from microarray analysis and applicable to RNA-seq inform this question. (a) It is necessary to normalize observations between samples to accommodate purely technical variation in overall patterns of expression. For example, samples provided to the sequencer have different amounts of DNA, resulting in variation in total numbers of sequenced and aligned reads independent of any difference in gene-level differential representation. This implies that the treatment should *affect only a fraction of the genes assayed*, otherwise treatment effects and protocol artifacts are confounded. (b) Between-gene measures of expression differ for reasons unrelated to levels of expression. For instance, standard protocols mean that a long gene is sequenced more often than a short gene, even when the number of mRNA molecules of the two genes are identical. This means that the most productive approach to differential representation will *compare genes across samples*, rather than compare levels of representation of different genes (gene set enrichment analysis and other approaches to between-gene comparison are statistically interesting in part because of the need to overcome between-gene differences arising for purely technical reasons). The combination of lessons (a) and (b) dictate that the treatment should affect only a subset of the genes under study, and that 'interesting' results correspond to treatment groups with differences at the gene level. *A priori* motivation, e.g., about well-defined pathways as targets of differential representation, may trump part (b) of this guideline.

4.3.2 Batch effects

The reality of executing designed experiments may mean that there are known but unavoidable factors that confound the analysis, but that are not of fundamental biological interest. Perhaps samples are being processed by different groups, or processing is spread over several months to accommodate personnel or sequencer availability. It is essential to avoid confounding such factors with biologically relevant parts of the experiment. Such batch effects are pervasive in high-throughput analysis of diverse data types [20]; addressing batch effects helps to reduce dependence, stabilize error rate estimates, and improve reproducibility.

Having acknowledged a potentially confounding factor, what is to be done? A first reaction might be randomization – arrange for samples to be processed in a random order, for instance, rather than by treatment group – but a better strategy is usually to include a blocking factor, e.g., processed by lab 'A' versus lab 'B' and to ensure that treatments are represented by replicates in each blocking factor. The down-stream analysis can then use replication to statistically accommodate such effects.

An alternative to explicitly modeling batch effects is to identify 'surrogate variables'. Surrogate variables are covariates constructed directly from the data, and can be used in subsequent analysis to adjust for unknown, un-modeled, or latent sources of noise [17, 18, 19]. The *sva* package implements surrogate variable analysis, and can be used with RNA-seq and many other high dimensional data types. *sva* estimates surrogate variables for inclusion in subsequent analysis, or removes known batch effects using ComBat [14].

An interesting approach to addressing batch effects in studies where new samples are accumulated incrementally (e.g., patient assays from physician offices) is to create a 'frozen' correction on a training data set, and perform per-sample correction on new samples as they become available. This is similar to the 'frozen' RMA approach to normalization developed by McCall et al., [24], and is implemented by the `fsva` function in the *sva* package.

4.3.3 Summarizing

The summary process tallies the number of reads aligning in each region (e.g., gene) of interest. The simplest method is to simply count reads overlapping each region, dividing by the length of the region of interest to accommodate differences in gene length. This is the 'RPKM' (reads per kilobase per million reads) of Mortazavi et al. [27]. One problem with this approach is that reads are not sampled uniformly across genes (Figure 4.1; [12]), so gene length (the 'PK' part of RPKM) is not a good proxy for expression level.

More fundamentally, each read represents an observation, and contributes to the certainty with which a gene is measured as 'expressed'. A summary measure like RPKM fails to incorporate uncertainty – a particular value of RPKM might result from alignment of one or 100 reads. This contrasts with a simple count of the number of reads in the region of interest. Furthermore, count data has known statistical properties that can be exploited in down-stream statistical analysis. Thus the result of summarization most useful for assessing differential expression is read count.

How to count? For instance, should a read that partly overlaps a 5' UTR or an intron be included in a tally? What about reads that overlap multiple genes? This is a non-trivial question because alignment is only approximate (reflecting sequencing and other biases) and because sample preparation protocols and organism biology (e.g., whether the UTR or fully mature mRNA is sequenced) may dictate particular counting strategies; more elaborate counting strategies might be entertained for paired end reads. Anders enumerates some counting strategies[1]; these are implemented in his *HTSeq* python scripts, in `summarizeOverlaps` in the *GenomicRanges* package, or in functions in the (linux-only) *Rsubread* and *gmapR* packages.

[1] `http://www-huber.embl.de/users/anders/HTSeq/doc/count.html`

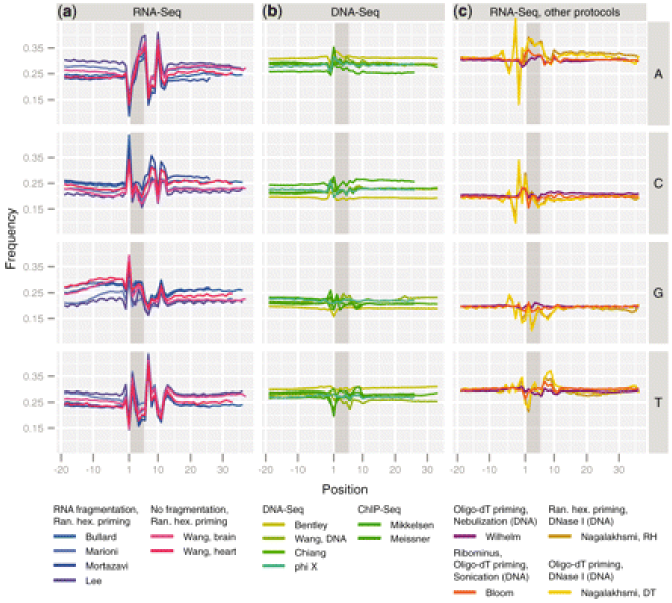


Figure 4.1: Nucleotide frequency versus position relative to start of alignment, various experiments and protocols; see [12].

4.3.4 Normalization

Normalization arises from the need to correct for purely technical differences between samples. The most common symptom of the need for normalization is differences in the total number of aligned reads. The ‘M’ part of RPKM measure mentioned in the context of summarization is one way of normalizing for total count. This normalization is not appropriate, because the distribution of aligned reads across genes within a sample is not uniform – some regions receive many more alignments than do others – and this distribution may differ between samples.

The overall strategy with normalization is to choose an appropriate baseline, and express sample counts relative to that baseline. There are several approaches to choice of appropriate baseline. One might choose total count for normalization, but this is a poor choice when one or a few regions of interest are very well represented – we are normalizing to the well-represented genes rather than to sequencing depth in each sample. Other straight-forward approaches include use of house-keeping genes, or the expression level from a particular quantile of the distribution of gene expression values of each sample [5]. One might attempt a robust estimate of sample abundance that is less sensitive to extreme outliers, e.g., the trimmed geometric mean of counts [1]. Another approach is TMM [31], which measures the trimmed mean of M and A values (M values are the log fold change in the number of reads aligning to a region of interest measured relative to an average or arbitrary sample, A is the average count of a gene; the trimmed mean discards regions of interest that have extreme M or A values and calculates the mean M value of the remainder); the inverse of this mean is used to weight samples. More data-driven approaches exploiting the gene-specific properties include conditional quantile normalization (implemented in the *cqn* package; [13]).

Another approach to normalization, increasingly popular as experiment size and data consistency increases, is to perform a data transformation and apply normalization methods developed for analysis of microarrays. Examples of this approach include `varianceStabilizingTransformation` from the *DESeq2* package, and `voom` from the *limma* package; see the corresponding help pages of these functions for details).

4.3.5 Error model

A Negative Binomial error model is often appropriate for ‘smaller’ experiments. These models combine Poisson (‘shot’ noise, i.e., within-sample techincal and sampling variation in read counts) with variation between biological samples. The *edgeR* [25] and *DESeq* [1] (now *DESeq2*) packages implement these models. Negative bionomial error models involve estimation of dispersion parameters, which are estimated poorly in small samples. *edgeR* and *DESeq2* adopt different data-driven approaches to arrive at more robust dispersion estimates; the packages, relying on different strategies to moderate per-gene estimates with more robust local estimates derived from genes with similar expression values. Other

Table 4.2: Selected *Bioconductor* packages for RNA-seq analysis.

Package	Description
EDASeq	Exploratory analysis and QA; also *qrqc*, *ShortRead*, *DESeq2*.
edgeR, *DESeq2*	Generalized Linear Models using negative binomial error.
BitSeq	Bayesian inference of individual transcript abundances followed by differential expression.
DEXSeq	Exon-level differential representation.
DSS, *vsn*, *cqn*	RNA-seq normalization methodologies. Also `voom` in *limma*.
goseq	Gene set enrichment tailored to RNAseq count data; also *limma*'s `roast` or `camera` after transformation with `voom`.
QuasR	Workflow.
Rsubread	Alignment (Linux only); also *gmapR*; *Biostrings* `matchPDict` for special-purpose alignments.
cummeRbund	Exploration and analysis of Cufflinks results.

approaches are possible; *DSS* [37] estimates are based on γ-Poisson or β-Binomial distributions.

As number of replicates become large, the importance of explicitly modeling biological sampling variance decrease. This encourages use of the Poisson-Tweedie family of distributions to model count data [9].

4.3.6 Multiple comparison

1. Increase statistical power and reduce false discovery rate by filtering regions of interest prior to analysis.
2. Motivation (a): just because genes are assayed does not mean, *a priori*, that they represent something requiring a statistical test. (b) Some observations, e.g., zero counts across all samples, cannot possibly be statistically significant, independent of hypothesis under investigation.
3. Approach – detection or 'K over A'-style filter; representation of a minimum of A (normalized) read counts in at least K samples. A usually measured as counts per million. Guidelines for choice of values a little *ad hoc*; see, e.g., the *edgeR* user manual. Variance filter, e.g., IQR (inter-quartile range) provides a robust estimate of variability; can be used to rank and discard least-varying regions.

4.4 Selected Bioconductor software for RNA-seq Analysis

Bioconductor packages play a role in several stages of an RNA-seq analysis (Table 4.2; a more comprehensive list is under the RNAseq and HighThroughputSequencing BiocViews terms). The *GenomicRanges* infrastructure can be effectively employed to quantify known exon or transcript abundances. Quantified abundances are in essence a matrix of counts, with rows representing features and columns samples. The *edgeR* [31] and *DESeq2* [1] packages facilitate analysis of this data in the context of designed experiments, and are appropriate when the questions of interest involve between-sample comparisons of relative abundance. The *DEXSeq* package extends the approach in *edgeR* and *DESeq2* to ask about within-gene, between group differences in exon use, i.e., for a given gene, do groups differ in their exon use?

Chapter 5

DESeq2 Work Flow Exercises

For this chapter, follow in-course instructions for working with the *DESeq2* vignette[1].

One strategy is:

1. Read section 4 to gain insight into the theoretical basis of the approach
2. Pursue section 1.2.3 to input data from a 'count' matrix, and description of column (sample) data. This is the most likely starting point for a typical analysis.
3. Note sections 1.2.5 and 1.2.6, but follow the main thread at section 1.3.

[1] `http://bioconductor.org/packages/devel/bioc/vignettes/DESeq2/inst/doc/DESeq2.pdf`

Chapter 6

Annotation and Visualization

6.1 Gene-centric Annotation

Bioconductor provides extensive annotation resources. These can be *gene*-, or *genome*-centric. Annotations can be provided in packages curated by *Bioconductor*, or obtained from web-based resources. Gene-centric *AnnotationDbi* packages include:

- Organism level: e.g. *org.Mm.eg.db*, *Homo.sapiens*.
- Platform level: e.g. *hgu133plus2.db*, *hgu133plus2.probes*, *hgu133plus2.cdf* .
- Homology level: e.g. *hom.Dm.inp.db*.
- System biology level: *GO.db*, *KEGG.db*, *Reactome.db*.

Examples of genome-centric packages include:

- *GenomicFeatures*, to represent genomic features, including constructing reproducible feature or transcript data bases from file or web resources.
- Pre-built transcriptome packages, e.g. *TxDb.Hsapiens.UCSC.hg19.knownGene* based on the *H. sapiens* UCSC hg19 knownGenes track.
- *BSgenome* for whole genome sequence representation and manipulation.
- Pre-built genomes, e.g., *BSgenome.Hsapiens.UCSC.hg19* based on the *H. sapiens* UCSC hg19 build.

Web-based resources include

- *biomaRt* to query biomart resource for genes, sequence, SNPs, and etc.
- *rtracklayer* for interfacing with browser tracks, especially the UCSC genome browser.

6.1.1 AnnotationDbi

Organism-level ('org') packages contain mappings between a central identifier (e.g., Entrez gene ids) and other identifiers (e.g. GenBank or Uniprot accession number, RefSeq id, etc.). The name of an org package is always of the form *org.<Sp>.<id>.db* (e.g. *org.Sc.sgd.db*) where *<Sp>* is a 2-letter abbreviation of the organism (e.g. `Sc` for *Saccharomyces cerevisiae*) and *<id>* is an abbreviation (in lower-case) describing the type of central identifier (e.g. `sgd` for gene identifiers assigned by the *Saccharomyces* Genome Database, or `eg` for Entrez gene ids). The "How to use the '.db' annotation packages" vignette in the *AnnotationDbi* package (org packages are only one type of ".db" annotation packages) is a key reference. The '.db' and most other *Bioconductor* annotation packages are updated every 6 months.

Annotation packages contain an object named after the package itself. These objects are collectively called *AnnotationDb* objects, with more specific classes named *OrgDb*, *ChipDb* or *TranscriptDb* objects. Methods that can be applied to these objects include `cols`, `keys`, `keytypes` and `select`. Common operations for retrieving annotations are summarized in Table 6.1.

Exercise 14

What is the name of the org package for Drosophila*? Load it. Display the OrgDb object for the org.Dm.eg.db package. Use the* `cols` *method to discover which sorts of annotations can be extracted from it.*

Use the `keys` *method to extract UNIPROT identifiers and then pass those keys in to the* `select` *method in such a way that you extract the SYMBOL (gene symbol) and KEGG pathway information for each.*

Table 6.1: Common operations for retrieving and manipulating annotations.

Category	Function	Description
Discover	`cols`	List the kinds of columns that can be returned
	`keytypes`	List columns that can be used as keys
	`keys`	List values that can be expected for a given keytype
	`select`	Retrieve annotations matching `keys`, `keytype` and `cols`
Manipulate	`setdiff`, `union`, `intersect`	Operations on sets
	`duplicated`, `unique`	Mark or remove duplicates
	`%in%`, `match`	Find matches
	`any`, `all`	Are any `TRUE`? Are all?
	`merge`	Combine two different `data.frames` based on shared keys
*GRanges**	`transcripts`, `exons`, `cds`	Features (transcripts, exons, coding sequence) as *GRanges*.
	`transcriptsBy` , `exonsBy` `cdsBy`	Features group by gene, transcript, etc., as *GRangesList*.

Use `select` to retrieve the ENTREZ and SYMBOL identifiers of all genes in the KEGG pathway 00310.

Solution: The *OrgDb* object is named `org.Dm.eg.db`.

```
> library(org.Dm.eg.db)
> columns(org.Dm.eg.db)
```

```
 [1] "ENTREZID"     "ACCNUM"       "ALIAS"        "CHR"          "CHRLOC"
 [6] "CHRLOCEND"    "ENZYME"       "MAP"          "PATH"         "PMID"
[11] "REFSEQ"       "SYMBOL"       "UNIGENE"      "ENSEMBL"      "ENSEMBLPROT"
[16] "ENSEMBLTRANS" "GENENAME"     "UNIPROT"      "GO"           "EVIDENCE"
[21] "ONTOLOGY"     "GOALL"        "EVIDENCEALL"  "ONTOLOGYALL"  "FLYBASE"
[26] "FLYBASECG"    "FLYBASEPROT"
```

```
> keytypes(org.Dm.eg.db)
```

```
 [1] "ENTREZID"     "ACCNUM"       "ALIAS"        "CHR"          "CHRLOC"
 [6] "CHRLOCEND"    "ENZYME"       "MAP"          "PATH"         "PMID"
[11] "REFSEQ"       "SYMBOL"       "UNIGENE"      "ENSEMBL"      "ENSEMBLPROT"
[16] "ENSEMBLTRANS" "GENENAME"     "UNIPROT"      "GO"           "EVIDENCE"
[21] "ONTOLOGY"     "GOALL"        "EVIDENCEALL"  "ONTOLOGYALL"  "FLYBASE"
[26] "FLYBASECG"    "FLYBASEPROT"
```

```
> uniprotKeys <- head(keys(org.Dm.eg.db, keytype="UNIPROT"))
> cols <- c("SYMBOL", "PATH")
> select(org.Dm.eg.db, keys=uniprotKeys, columns=cols, keytype="UNIPROT")
```

```
  UNIPROT  SYMBOL  PATH
1  Q8IRZ0  CG3038  <NA>
2  Q95RP8  CG3038  <NA>
3  M9PGH7     G9a 00310
4  Q95RU8     G9a 00310
5  Q9W5H1 CG13377  <NA>
6  P39205     cin  <NA>
```

Selecting UNIPROT and SYMBOL ids of KEGG pathway 00310 is very similar:

```
> kegg <- select(org.Dm.eg.db, "00310", c("UNIPROT", "SYMBOL"), "PATH")
> nrow(kegg)
```

```
[1] 35
```

```
> head(kegg, 3)

   PATH UNIPROT  SYMBOL
1 00310  M9PGH7     G9a
2 00310  Q95RU8     G9a
3 00310  M9NE25 Hmt4-20
```

Exercise 15
For convenience, `lrTest`, a DGEGLM object from the RNA-seq chapter, is included in the SequenceAnalysisData package. The following code loads this data and creates a 'top table' of the ten most differentially represented genes. This top table is then coerced to a `data.frame`.

```
> library(SequenceAnalysisData)
> library(edgeR)
> library(org.Dm.eg.db)
> data(lrTest)
> tt <- as.data.frame(topTags(lrTest))
```

Extract the Flybase gene identifiers (`FLYBASE`) from the row names of this table and map them to their corresponding Entrez gene (`ENTREZID`) and symbol ids (`SYMBOL`) using `select`. Use `merge` to add the results of `select` to the top table.

Solution:

```
> fbids <- rownames(tt)
> cols <- c("ENTREZID", "SYMBOL")
> anno <- select(org.Dm.eg.db, fbids, cols, "FLYBASE")
> ttanno <- merge(tt, anno, by.x=0, by.y="FLYBASE")
> dim(ttanno)

[1] 10  8

> head(ttanno, 3)

    Row.names logConc logFC LR.statistic  PValue     FDR ENTREZID  SYMBOL
1 FBgn0000071     -11   2.8          183 1.1e-41 1.1e-38    40831     Ama
2 FBgn0024288     -12  -4.7          179 7.1e-41 6.3e-38    45039 Sox100B
3 FBgn0033764     -12   3.5          188 6.8e-43 7.8e-40     <NA>    <NA>
```

6.1.2 biomaRt and other web-based resources

A short summary of select *Bioconductor* packages enabling web-based queries is in Table 6.2.

Using biomaRt The *biomaRt* package offers access to the online biomart resource. this consists of several data base resources, referred to as 'marts'. Each mart allows access to multiple data sets; the *biomaRt* package provides methods for mart and data set discovery, and a standard method `getBM` to retrieve data.

Exercise 16
Load the biomaRt package and list the available marts. Choose the ensembl *mart and list the datasets for that mart. Set up a mart to use the* ensembl *mart and the* hsapiens_gene_ensembl *dataset.*

A biomaRt dataset can be accessed via getBM. In addition to the mart to be accessed, this function takes filters and attributes as arguments. Use `filterOptions` and `listAttributes` to discover values for these arguments. Call `getBM` using filters and attributes of your choosing.

Table 6.2: Selected packages querying web-based annotation services.

Package	Description
AnnotationHub	Ensembl, Encode, dbSNP, UCSC data objects
biomaRt	`http://biomart.org`, Ensembl and other annotations
uniprot.ws	`http://uniprot.org`, protein annotations
KEGGREST	`http://www.genome.jp/kegg`, KEGG pathways
SRAdb	`http://www.ncbi.nlm.nih.gov/sra`, sequencing experiments.
rtracklayer	`http://genome.ucsc.edu`, genome tracks.
GEOquery	`http://www.ncbi.nlm.nih.gov/geo/`, array and other data
ArrayExpress	`http://www.ebi.ac.uk/arrayexpress/`, array and other data

Solution:

```
> library(biomaRt)
> head(listMarts(), 3)                      ## list the marts
> head(listDatasets(useMart("ensembl")), 3) ## mart datasets
> ensembl <-                                ## fully specified mart
+     useMart("ensembl", dataset = "hsapiens_gene_ensembl")
> head(listFilters(ensembl), 3)             ## filters
> myFilter <- "chromosome_name"
> head(filterOptions(myFilter, ensembl), 3) ## return values
> myValues <- c("21", "22")
> head(listAttributes(ensembl), 3)          ## attributes
> myAttributes <- c("ensembl_gene_id","chromosome_name")
> ## assemble and query the mart
> res <- getBM(attributes =  myAttributes, filters =  myFilter,
+              values =  myValues, mart = ensembl)
```

Use `head(res)` to see the results.

6.2 Genomic Annotation

6.2.1 AnnotationHub

6.2.2 Whole genome sequences

There are a diversity of packages and classes available for representing large genomes. Several include:

TxDb.* For transcript and other genome / coordinate annotation.

BSgenome For whole-genome representation. See `available.packages` for pre-packaged genomes, and the vignette 'How to forge a BSgenome data package' in the

Homo.sapiens For integrating *TxDb** and *org.** packages.

SNPlocs.* For model organism SNP locations derived from dbSNP.

`FaFile` (*Rsamtools*) for accessing indexed FASTA files.

SIFT.*, PolyPhen Variant effect scores.

6.2.3 Gene models: TxDb.* packages for model organisms

Genome-centric packages are very useful for annotations involving genomic coordinates. It is straight-forward, for instance, to discover the coordinates of coding sequences in regions of interest, and from these retrieve corresponding DNA or protein coding sequences. Other examples of the types of operations that are easy to perform with genome-centric annotations include defining regions of interest for counting aligned reads in RNA-seq experiments and retrieving DNA sequences underlying regions of interest in ChIP-seq analysis, e.g., for motif characterization.

Exercise 17

Load the 'transcript.db' package relevant to the dm3 build of D. melanogaster*. Use* `select` *and friends to select the Flybase gene ids of the top table* `tt` *and the Flybase transcript names (TXNAME) and Entrez gene identifiers (GENEID).*

Use `cdsBy` *to extract all coding sequences, grouped by transcript. Subset the coding sequences to contain just the transcripts relevant to the top table. How many transcripts are there? What is the structure of the first transcript's coding sequence?*

Load the 'BSgenome' package for the dm3 build of D. melanogaster*. Use the coding sequences ranges of the previous part of this exercise to extract the underlying DNA sequence, using the* `extractTranscriptsFromGenome` *function. Use Biostrings'* `translate` *to convert DNA to amino acid sequences.*

Solution: The following loads the relevant Transcript.db package, and creates a more convenient alias to the *TranscriptDb* instance defined in the package.

```
> library(TxDb.Dmelanogaster.UCSC.dm3.ensGene)
> txdb <- TxDb.Dmelanogaster.UCSC.dm3.ensGene
```

We also need the data – flybase IDs from our differential expression analysis.

```
> library(SequenceAnalysisData)
> data(lrTest)
> fbids <- rownames(topTags(lrTest))
```

We can discover available keys (using `keys`) and columns (`cols`) in `txdb`, and then use `select` to retrieve the transcripts associated with each differentially expressed gene. The mapping between gene and transcript is not one-to-one – some genes have more than one transcript.

```
> txnm <- select(txdb, fbids, "TXNAME", "GENEID")
> nrow(txnm)

[1] 14

> head(txnm, 3)

       GENEID      TXNAME
1 FBgn0039155 FBtr0084549
2 FBgn0039827 FBtr0085755
3 FBgn0039827 FBtr0085756
```

The *TranscriptDb* instances can be queried for data that is more structured than simple data frames, and in particular return *GRanges* or *GRangesList* instances to represent genomic coordinates. These queries are performed using `cdsBy` (coding sequence), `transcriptsBy` (transcripts), etc., where a function argument `by` specifies how coding sequences or transcripts are grouped. Here we extract the coding sequences grouped by transcript, returning the transcript names, and subset the resulting *GRangesList* to contain just the transcripts of interest to us. The first transcript is composed of 6 distinct coding sequence regions.

```
> txnm <- txnm[!is.na(txnm$TXNAME),,drop=FALSE]
> cds <- cdsBy(txdb, "tx", use.names=TRUE)[txnm$TXNAME]
> length(cds)

[1] 13

> cds[1]

GRangesList of length 1:
$FBtr0084549
GRanges with 6 ranges and 3 metadata columns:
      seqnames               ranges strand |    cds_id    cds_name exon_rank
         <Rle>            <IRanges>  <Rle> | <integer> <character> <integer>
  [1]    chr3R [19970946, 19971592]      + |     41058        <NA>         2
  [2]    chr3R [19971652, 19971770]      + |     41059        <NA>         3
  [3]    chr3R [19971831, 19972024]      + |     41060        <NA>         4
  [4]    chr3R [19972088, 19972461]      + |     41061        <NA>         5
  [5]    chr3R [19972523, 19972589]      + |     41062        <NA>         6
  [6]    chr3R [19972918, 19973094]      + |     41063        <NA>         7

---
seqlengths:
     chr2L     chr2R     chr3L     chr3R ...   chrXHet   chrYHet chrUextra
  23011544  21146708  24543557  27905053 ...    204112    347038  29004656
```

The following code loads the appropriate BSgenome package; the `Dmelanogaster` object refers to the whole genome sequence represented in this package. The remaining steps extract the DNA sequence of each transcript, and translates these to amino acid sequences. Issues of strand are handled correctly.

```
> library(BSgenome.Dmelanogaster.UCSC.dm3)
> txx <- extractTranscriptsFromGenome(Dmelanogaster, cds)
> length(txx)

[1] 13

> head(txx, 3)

  A DNAStringSet instance of length 3
    width seq                                               names
[1]  1578 ATGGGCAGCATGCAAGTGGCGCT...TGCAGATCAAGTGCAGCGACTAG FBtr0084549
[2]  2760 ATGCTGCGTTATCTGGCGCTTTC...TTGCTGCCCCATTCGAACTTTAG FBtr0085755
[3]  2217 ATGGCACTCAAGTTTCCCACAGT...TTGCTGCCCCATTCGAACTTTAG FBtr0085756

> head(translate(txx), 3)

  A AAStringSet instance of length 3
    width seq
[1]   526 MGSMQVALLALLVLGQLFPSAVANGSSSYSSTST...VLDDSRNVFTFTTPKCENFRKRFPKLQIKCSD*
[2]   920 MLRYLALSEAGIAKLPRPQSRCYHSEKGVWGYKP...YCGRCEAPTPATGIGKVHKREVDEIVAAPFEL*
[3]   739 MALKFPTVKRYGGEGAESMLAFFWQLLRDSVQAN...YCGRCEAPTPATGIGKVHKREVDEIVAAPFEL*
```

6.3 Visualizing Sequence Data

R has some great visualization packages; essential references include [7] for a general introduction, Murrell [28] for base graphics, Sarkar [34] for *lattice*, and Wickham [36] for *ggplot2*. Here we take a quick tour of visualization facilities tailed for sequence data and using *Bioconductor* approaches.

6.3.1 Gviz

The *Gviz* package produces very elegant data organized in a more-or-less familiar 'track' format. The following exercises walk through the *Gviz* User guide Section 2.

Exercise 18
Load the Gviz package and sample GRanges containing genomic coordinates of CpG islands. Create a couple of variables with information on the chromosome and genome of the data (how can this information be extracted from the `cpgIslands` *object?).*

```
> library(Gviz)
> data(cpgIslands)
> chr <- "chr7"
> genome <- "hg19"
```

The basic idea is to create a track, perhaps with additional attributes, and to plot it. There are different types of track, and we create these one at a time. We start with a simple annotation track

```
> atrack <- AnnotationTrack(cpgIslands, name="CpG")
> plotTracks(atrack)
```

Then add a track that represents genomic coordinates. Tracks are combined when plotted, as a simple list. The vertical ordering of tracks is determined by their position in the list.

```
> gtrack <- GenomeAxisTrack()
> plotTracks(list(gtrack, atrack))
```

We can add an ideogram to provide overall orientation...

```
> itrack <- IdeogramTrack(genome=genome, chromosome=chr)
> plotTracks(list(itrack, gtrack, atrack))
```

and a more elaborate gene model, as an *data.frame* or *GRanges* object with specific columns of metadata.

```
> data(geneModels)
> grtrack <-
+     GeneRegionTrack(geneModels, genome=genome,
+                     chromosome=chr, name="Gene Model")
> tracks <- list(itrack, gtrack, atrack, grtrack)
> plotTracks(tracks)
```

Zooming out changes the location box on the ideogram

```
> plotTracks(tracks, from=2.5e7, to=2.8e7)
```

When zoomed in we can add sequence data

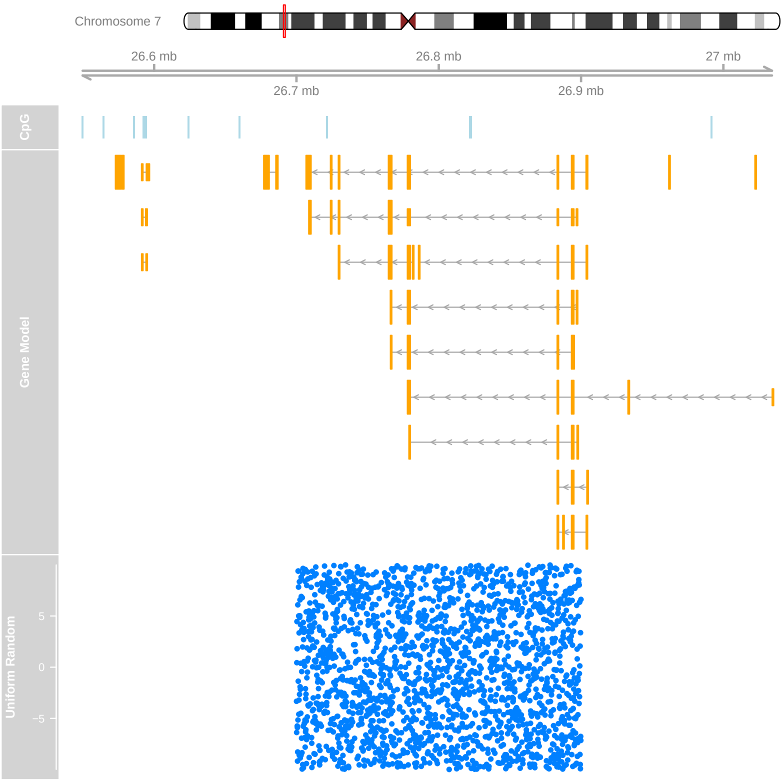
```
> library(BSgenome.Hsapiens.UCSC.hg19)
> strack <- SequenceTrack(Hsapiens, chromosome=chr)
> plotTracks(c(tracks, strack), from=26450430, to=26450490, cex=.8)
```

As the *Gviz* vignette humbly says, ‘so far we have replicated the features of a whole bunch of other genome browser tools out there‘. We'd like to be able integrate our data into these plots, with a rich range of plotting options. The key is the DataTrack function, which we demonstrate with some simulated data; this final result is shown in Figure 6.1.

```
> ## some data
> lim <- c(26700000, 26900000)
> coords <- seq(lim[1], lim[2], 101)
> dat <- runif(length(coords) - 1, min=-10, max=10)
> ## DataTrack
> dtrack <-
+     DataTrack(data=dat, start=coords[-length(coords)],
+               end= coords[-1], chromosome=chr, genome=genome,
+               name="Uniform Random")
> plotTracks(c(tracks, dtrack))
```

Section 4.3 of the *Gviz* vignette illustrates flexibility of the data track.

Figure 6.1: *Gviz* ideogram, genome coordinate, annotation, and data tracks.



6.3.2 shiny for easy interactive reports

As a final example of visualization, the *shiny* package and web site [1] offers a fun and comparatively easy way to develop interactive, browser-based visualizations. These visualizations are an excellent way to provide sophisticated exploratory or summary analysis in a very accessible way. The idea is to write a 'user interface' component that describes how a page is to be presented to users, and a 'server' that describes how the data are to be calculated or modified in responses to user choices. The programming model is 'reactive', where changes in a user choice automatically trigger re-calculations in the server. This reactive model is like in a spreadsheet with a formula, where adjusting a cell that the formula references triggers re-calculation of the formula. Just like in a spreadsheet, someone creating a *shiny* application does not have to work hard to make reactivity work.

A simple *shiny* application is at

```
> library(shiny)
> appDir <- system.file("shiny",
+     "AnnotationTable", package="StatisticalComputing2013")
```

The application provides a simple way to to select annotations from different *org.** packages. Take the application for a spin:

```
> runApp(appDir)
```

The application consists of three files. `ui.R` defines the user interface, how various widgets appear in a web browser. This is the code that a 'client' would need to be able to use the application. View the content of this file with

```
> noquote(readLines(file.path(appDir, "ui.R")))
```

[1] `http://www.rstudio.com/shiny/`

```
shinyUI(pageWithSidebar(

    headerPanel("Annotation"),

    sidebarPanel(

        textInput("keys", "ENTREZ identifiers"),

        selectInput("organism", "Organism",
                    choices=names(map)),

        selectInput("columns", "Selected annotations",
                    choices=columns(map[[ "Human" ]]),
                    selected=c("SYMBOL", "GENENAME"),
                    multiple=TRUE)

        ),

    mainPanel(

        h4("Results (maximum 10 identifiers)"),
        tableOutput("view")

        )

    ))
```

`server.R` defines the computations that are performed in response to user selections. `global.R` defines global variables needed for both the user interface and server.

Exercise 19
(For technical reasons, this cannot easily be done on the AMI). Start R on your local computer, and install shiny

```
> source("http://bioconductor.org/biocLite.R")
> biocLite("shiny")
```

Download the `ui.R`, `server.R`, and `global.R` files from the AMI to a directory on your local computer, and run the app using `runApp` as above.

Now try making a simple change to the `ui.R` file, e.g., changing the name of the `headerPanel` from `Annotation` to `My Annotations`. Return to the browser and, without *restarting the shiny appilcation, reload the page. Note that your change is incorporated!*

Try introducing an error, e.g,. 'forgetting' to include the comma after the `headerPanel` function. Reload the app to see how shiny responds.

Implement a feature of your choice in the ui or server.

References

[1] S. Anders and W. Huber. Differential expression analysis for sequence count data. *Genome Biol*, 11(10):R106, 2010.

[2] S. Anders, D. J. McCarthy, Y. Chen, M. Okoniewski, G. K. Smyth, W. Huber, and M. D. Robinson. Count-based differential expression analysis of RNA sequencing data using R and Bioconductor. *Nat Protoc*, 8(9):1765–1786, Sep 2013.

[3] D. Bentley, S. Balasubramanian, H. Swerdlow, G. Smith, J. Milton, C. Brown, K. Hall, D. Evers, C. Barnes, H. Bignell, et al. Accurate whole human genome sequencing using reversible terminator chemistry. *Nature*, 456(7218):53–59, 2008.

[4] A. N. Brooks, L. Yang, M. O. Duff, K. D. Hansen, J. W. Park, S. Dudoit, S. E. Brenner, and B. R. Graveley. Conservation of an RNA regulatory map between Drosophila and mammals. *Genome Research*, pages 193–202, 2011.

[5] J. H. Bullard, E. Purdom, K. D. Hansen, and S. Dudoit. Evaluation of statistical methods for normalization and differential expression in mrna-seq experiments. *BMC bioinformatics*, 11(1):94, 2010.

[6] P. Burns. The R inferno. Technical report, 2011.

[7] W. Chang. *R Graphics Cookbook*. O'Reilly Media, Incorporated, 2012.

[8] P. Dalgaard. *Introductory Statistics with R*. Springer, 2nd edition, 2008.

[9] M. Esnaola, P. Puig, D. Gonzalez, R. Castelo, and J. R. Gonzalez. A flexible count data model to fit the wide diversity of expression profiles arising from extensively replicated rna-seq experiments. *BMC Bioinformatics*, 14:254, 2013.

[10] R. Gentleman. *R Programming for Bioinformatics*. Computer Science & Data Analysis. Chapman & Hall/CRC, Boca Raton, FL, 2008.

[11] F. Hahne, W. Huber, R. Gentleman, and S. Falcon. *Bioconductor case studies*. Springer, 2008.

[12] K. D. Hansen, S. E. Brenner, and S. Dudoit. Biases in illumina transcriptome sequencing caused by random hexamer priming. *Nucleic acids research*, 38(12):e131–e131, 2010.

[13] K. D. Hansen, R. A. Irizarry, and Z. Wu. Removing technical variability in RNA-seq data using conditional quantile normalization. *Biostatistics*, 13(2):204–216, 2012.

[14] W. Johnson, C. Li, and A. Rabinovic. Adjusting batch effects in microarray data using empirical bayes methods. *Biostatistics*, 8(1):118–127, 2007.

[15] B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biol.*, 10:R25, 2009.

[16] M. Lawrence, W. Huber, H. PagÃĺs, P. Aboyoun, M. Carlson, R. Gentleman, M. T. Morgan, and V. J. Carey. Software for computing and annotating genomic ranges. *PLoS Comput Biol*, 9(8):e1003118, 08 2013.

[17] J. Leek and J. Storey. Capturing heterogeneity in gene expression studies by 'surrogate variable analysis'. *PLoS Genetics 3:e161*, 2007.

[18] J. Leek and J. Storey. A general framework for multiple testing dependence. *Proceedings of the National Academy of Sciences 105:18718-18723*, 2008.

[19] J. T. Leek. Asymptotic conditional singular value decomposition for high-dimensional genomic data. *Biometrics*, 67:344–352, Jun 2011.

[20] J. T. Leek, R. B. Scharpf, H. C. Bravo, D. Simcha, B. Langmead, W. E. Johnson, D. Geman, K. Baggerly, and R. A. Irizarry. Tackling the widespread and critical impact of batch effects in high-throughput data. *Nat. Rev. Genet.*, 11:733–739, Oct 2010.

[21] H. Li and R. Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, 26:589–595, Mar 2010.

[22] J. C. Marioni, C. E. Mason, S. M. Mane, M. Stephens, and Y. Gilad. Rna-seq: an assessment of technical reproducibility and comparison with gene expression arrays. *Genome research*, 18(9):1509–1517, 2008.

[23] N. Matloff. *The Art of R Programming*. No Starch Pess, 2011.

[24] M. N. McCall, B. M. Bolstad, and R. A. Irizarry. Frozen robust multiarray analysis (frma). *Biostatistics*, 11(2):242–253, 2010.

[25] McCarthy, D. J., Chen, Yunshun, Smyth, and G. K. Differential expression analysis of multifactor rna-seq experiments with respect to biological variation. *Nucleic Acids Research*, 40(10):–9, 2012.

[26] J. Meys and A. de Vries. *R For Dummies*. For Dummies, 2012.

[27] A. Mortazavi, B. A. Williams, K. McCue, L. Schaeffer, and B. Wold. Mapping and quantifying mammalian transcriptomes by rna-seq. *Nature methods*, 5(7):621–628, 2008.

[28] P. Murrell. *R graphics*. Chapman & Hall/CRC, 2005.

[29] P. J. Park. ChIP-seq: advantages and challenges of a maturing technology. *Nat. Rev. Genet.*, 10:669–680, Oct 2009. [PubMed Central:PMC3191340] [DOI:10.1038/nrg2641] [PubMed:19736561].

[30] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2013. ISBN 3-900051-07-0.

[31] M. D. Robinson, D. J. McCarthy, and G. K. Smyth. edgeR: a Bioconductor package for differential expression analysis of digital gene expression data. *Bioinformatics*, 26:139–140, Jan 2010.

[32] C. Ross-Innes, R. Stark, A. Teschendorff, K. Holmes, H. Ali, M. Dunning, G. Brown, O. Gojis, I. Ellis, A. Green, et al. Differential oestrogen receptor binding is associated with clinical outcome in breast cancer. *Nature*, 481(7381):389–393, 2012.

[33] J. Rothberg and J. Leamon. The development and impact of 454 sequencing. *Nature biotechnology*, 26(10):1117–1124, 2008.

[34] D. Sarkar. *Lattice: multivariate data visualization with R*. Springer, 2008.

[35] A. A. Shabalin. Matrix eqtl: ultra fast eqtl analysis via large matrix operations. *Bioinformatics*, 28(10):1353–1358, 2012.

[36] H. Wickham. *ggplot2: elegant graphics for data analysis*. Springer Publishing Company, Incorporated, 2009.

[37] H. Wu, C. Wang, and Z. Wu. A new shrinkage estimator for dispersion improves differential expression detection in rna-seq data. *Biostatistics*, 2012.